NEON crypto

Daniel J. Bernstein[1] and Peter Schwabe[2]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Institute of Information Science
Academia Sinica, 128 Section 2 Academia Road, Taipei 115-29, Taiwan
peter@cryptojedi.org

Abstract. NEON is a vector instruction set included in a large fraction of new ARM-based tablets and smartphones. This paper shows that NEON supports high-security cryptography at surprisingly high speeds; normally data arrives at lower speeds, giving the CPU time to handle tasks other than cryptography. In particular, this paper explains how to use a single 800MHz Cortex A8 core to compute the existing NaCl suite of high-security cryptographic primitives at the following speeds: 5.60 cycles per byte (1.14 Gbps) to encrypt using a shared secret key, 2.30 cycles per byte (2.78 Gbps) to authenticate using a shared secret key, 527102 cycles (1517/second) to compute a shared secret key for a new public key, 650102 cycles (1230/second) to verify a signature, and 368212 cycles (2172/second) to sign a message. These speeds make no use of secret branches and no use of secret memory addresses.

Keywords: vectorization-friendly cryptographic primitives, efficient software implementations, smartphones, tablets, there be dragons

1 Introduction

The Apple A4 CPU used in the iPad 1 (2010, 1GHz) and iPhone 4 (2010, 1GHz) contains a single Cortex A8 CPU core. The same CPU core also appears in many other tablets and smartphones. The point of this paper is that the Cortex A8 achieves impressive speeds for high-security cryptography:

- 5.60 cycles per byte to encrypt a message using a shared secret key;
- 2.30 cycles per byte to authenticate a message using a shared secret key;
- 527102 cycles to compute a shared secret key for a new public key;
- 650102 cycles to verify a signature on a short message; and
- 368212 cycles to sign a short message.

We do not claim that *all* high-security cryptographic primitives run well on the Cortex A8. Quite the opposite: we rely critically on a synergy between

- the capabilities of the "NEON" vector unit in the Cortex A8 and
- the parallelizability of some carefully selected cryptographic primitives.

The primitives we use are Salsa20 [9], a member of the final portfolio from the ECRYPT Stream Cipher Project; Poly1305 [5], a polynomial-evaluation message-authentication code similar to the message-authentication code in GCM; Curve25519 [6], an elliptic-curve Diffie–Hellman system; and Ed25519 [10], an elliptic-curve signature system that was introduced at CHES 2011. The rest of this paper explains how we use NEON to obtain such high speeds for these primitives.

It is not a coincidence that our selection matches the default primitives in NaCl, the existing "Networking and Cryptography library" [13] used in applications such as DNSCrypt [45]; vectorizability was one of the design criteria for NaCl. It is nevertheless surprising that a rather small vector unit, carrying out just one arithmetic instruction per cycle, can run these primitives at the speeds listed above. A high-power Intel Core 2 CPU core (at 45nm, like the Apple A4), with a 64-bit instruction set and three full 128-bit vector units, has cycle counts of 3.98/byte, 3.32/byte, 307053, 365742, and 106542 for the same five tasks with the best reported assembly-language

This work was supported by the National Science Foundation under grant 1018836; by the US Air Force grant AOARD-11-4092; and by the Career Development Award of the second author's employer. Permanent ID of this document: 9b53e3cd38944dcc8baf4753eeb1c5e7. Date: 2012.03.20.

implementations of the same primitives in the SUPERCOP benchmarking suite [12]; the Cortex A8 ends up much more competitive than one might expect. We also do better than the 697080 Cell cycles for Curve25519 achieved in [17], even though the Cell has more powerful permutation instructions and many more registers.

Side channels. All memory addresses and branch conditions in our software are public, depending only on message lengths. There is no data flow from secret data (keys, plaintext, etc.) to cache timing, branch timing, etc. We do not claim that our software is immune to *hardware* side-channel attacks such as power analysis, but we do claim that it is immune to *software* side-channel attacks such as [44], [2], and [47].

Benchmarking platform. The speeds reported above were measured on a low-cost Hercules eCAFE netbook (released and purchased in 2011) containing a Freescale i.MX515 CPU. This CPU has a single 800MHz Cortex A8 core. Occasionally we make comparisons to benchmarks that use OpenSSL or a C compiler; the netbook is shipped with Ubuntu 10.04, and in particular OpenSSL 0.9.8k and `gcc` 4.4.3, neither of which we claim is optimal.

All of our software has been checked against standard test suites. We are placing our software online to maximize verifiability of our results, and are placing it into the public domain to maximize reusability. Some of our preliminary results are already online and included in various public benchmark reports, but this paper is our first formal announcement and achieves even better speeds.

More CPUs with NEON. The Cortex A8 is not the only hardware design supporting the NEON instruction set. The Apple A5 CPU used in the iPad 2 (2011, 1GHz) and iPhone 4S (2011, 800MHz) contains two Cortex A9 CPU cores with NEON units. The NVIDIA Tegra 3 CPU used in the 2011 Asus Eee Pad Transformer Prime tablet (2011, 1.3GHz) and HTC One X smartphone (2012, 1.5GHz) contains four Cortex A9 CPU cores with NEON units. Qualcomm's "Snapdragon" series of CPUs reportedly includes a different NEON microarchitecture for the older "Scorpion" cores and a faster NEON microarchitecture for the newer "Krait" cores.

We have very recently benchmarked our software on a Scorpion, obtaining cycle counts of 5.40/byte, 1.89/byte, 606824, 756795, and 511123 for the five tasks listed above. We expect that further optimization for Cortex A9 and Snapdragon will produce even better results. The rest of this paper focuses on the original Cortex A8 NEON microarchitecture.

One should not think that *all* tablets and smartphones support NEON instructions. For example, NVIDIA omitted NEON from the Cortex A9 cores in the Tegra 2; lower-cost ARM11 processors do not support NEON and continue to appear in new devices; and some devices use Intel processors with a quite different instruction set. However, Apple alone has sold more than 50 million tablets with NEON and many more smartphones with NEON, and our sampling suggests that NEON also appears in the majority of new tablets and smartphones from other manufacturers. This paper turns all of these devices into powerful cryptographic engines, capable of protecting large volumes of data while leaving the CPU with enough time to actually do something useful with that data.

2 NEON instructions and speeds

This section reviews NEON's capabilities. This is not a comprehensive review: it focuses on the most important instructions for our software, and the main bottlenecks in those instructions. All comments about speed refer to the NEON unit in a single Cortex A8 core.

Registers. The NEON architecture has 16 128-bit vector registers (2048 bits overall), `q0` through `q15`. It also has 32 64-bit vector registers, `d0` through `d31`, but these registers share physical space with the 128-bit vector registers: `q0` is the concatenation of `d0` and `d1`, `q1` is the concatenation of `d2` and `d3`, etc.

For comparison, the basic ARM architecture has only 16 32-bit registers, `r0` through `r15`. Register `r13` is the stack pointer and register `r15` is the program counter, leaving only 14 32-bit registers (448 bits overall) for general use. One of the most obvious benefits of NEON for cryptography is

that it provides much more space in registers, reducing the number of loads and stores that we need.

Syntax. We rarely look at NEON register names, even though we write code in assembly: we use a higher-level assembly syntax that allows any number of names for 128-bit vector registers. For example, we write

```
diag3 ^= b0
```

and then an automatic translator produces traditional assembly language

```
veor q6,q6,q14
```

for assembly by the standard GNU assembler `gas`; here the translator has selected `q6` for `diag3` and `q14` for `b0`. We nevertheless pay close attention to the number of "live" 128-bit registers at each moment, reorganizing our computations to fit reasonably large amounts of work into registers.

The syntax is our own design. To build the translator we reused the existing `qhasm` toolkit [7] and wrote a short ARM+NEON machine-description file for `qhasm`. This file contains, for example, the line

```
4x r=s+t:>r=reg128:<s=reg128:<t=reg128:asm/vadd.i32 >r,<s,<t:
```

stating our syntax and the `gas` assembly-language syntax for a 4-way vectorized 32-bit addition, and also identifying the inputs and outputs of the instruction for the `qhasm` register allocator. The code examples in the rest of this paper use our syntax for the sake of readability; we do not assume that readers are already familiar with NEON.

We have also experimented extensively with writing NEON code in C, using compiler extensions for NEON instructions. However, we have found that assembly language gives us far better tradeoffs between software speed and programming effort. Assembly language has a reputation for being hard to read and write, but typical code such as

```
4x a0 = diag1 + diag0
4x b0 = a0 << 7
```

in our assembly-language syntax is as straightforward as

```
a0 = diag1 + diag0;
b0 = vshlq_n_u32(a0,7);
```

in C. The critical advantage of assembly language is that it provides more control. We frequently find that every available C compiler produces poorly scheduled code, leaving the NEON unit mostly idle; changing the C code to produce better assembly-language scheduling is a hit-and-miss affair, and it is also not clear how the compiler could be modified to do better, since the C language provides no way to express instruction priorities. Writing directly in assembly language eliminates this difficulty, allowing us to focus on higher-level questions of how to decompose larger computations (such as multiplications modulo $2^{255} - 19$) into pieces suitable for vectorization.

Arithmetic instructions. The Cortex A8 NEON microarchitecture has one 128-bit arithmetic unit. A typical arithmetic instruction such as

```
4x a = b + c
```

occupies the NEON arithmetic unit for one cycle. This instruction partitions the 128-bit output register `a` into four 32-bit quantities `a[0]`, `a[1]`, `a[2]`, `a[3]`, similarly partitions `b` and `c`, and then has the same effect as

```
a[0] = b[0] + c[0]
a[1] = b[1] + c[1]
a[2] = b[2] + c[2]
a[3] = b[3] + c[3]
```

where as usual `+` means addition modulo 2^{32}. Readers accustomed to two-operand architectures should note that there is no requirement to split this instruction into a copy `a = b` followed by `4x a += c`.

This instruction passes through several single-cycle NEON pipeline stages N1, N2, etc. It reads its input when it is in stage N2; if the input will not be ready then it already predicts the problem at the beginning of the pipeline and stalls there, also stalling subsequent NEON instructions. It makes its output available in stage N4, two cycles after reading the input, so another addition instruction that begins two cycles later (reaching N2 when the first instruction reaches N4) can read the output without stalling.

We comment that "addition has 2-cycle latency" would be an oversimplification, for reasons that will be clear in the next paragraph. We also warn readers that ARM's Cortex A8 manual [3] reports stage N3 for the output, even though an addition that begins the next cycle will in fact stall. This is not an isolated error in the manual, but rather an unusual convention for reporting output availability: ARM consistently lists the stage just *before* the output is ready. An online Cortex A8 cycle counter by Sobole [40] correctly displays this latency, although we encountered some other cases where it was too pessimistic.

A logical instruction such as

```
a = b ^ c
```

has the same performance as an addition. A subtraction instruction

```
4x a = b - c
```

occupies the arithmetic unit for one cycle, just like addition, but needs the `c` input one cycle earlier, in stage N1. Addition and subtraction thus each have latency 2 as input to an addition or to the positive part of a subtraction, but latency 3 as input to the negative part of a subtraction.

Shifting by a fixed distance is like subtraction in that it needs input in stage N1 and generates output in stage N4. NEON can combine three instructions for rotation into two instructions—

```
4x a = b << 7
4x a insert= b >> 25
```

—but the second instruction occupies the arithmetic unit for two cycles and generally causes larger latency problems than a separate shift and xor.

A pair of 32-bit multiplications, each producing a 64-bit result, uses one instruction:

```
c[0,1] = a[0] signed* b[0]; c[2,3] = a[1] signed* b[1]
```

This instruction occupies the arithmetic unit for two cycles, for a total throughput of one $32 \times 32 \to 64$-bit multiplication per cycle. This instruction reads `b` in stage N1, reads `a` in stage N2, and makes `c` available in stage N8. This instruction has a multiply-accumulate variant, carrying out additions for free:

```
c[0,1] += a[0] signed* b[0]; c[2,3] += a[1] signed* b[1]
```

The accumulator is normally read in stage N3, but is read much later *if* it is the result of a similar multiplication instruction. A typical sequence such as

```
c[0,1] = a[0] unsigned* b[0]; c[2,3] = a[1] unsigned* b[1]
c[0,1] += e[2] unsigned* f[2]; c[2,3] += e[3] unsigned* f[3]
c[0,1] += g[0] unsigned* h[2]; c[2,3] += g[1] unsigned* h[3]
```

takes six cycles without any stalls.

Loads, stores, and permutations. There is a 128-bit NEON load/store unit that runs in parallel with the NEON arithmetic unit. An aligned 128-bit or aligned 64-bit load or store consumes the load/store unit for one cycle and makes its result available in N2. Alignment is static (encoded explicitly in the instruction), not dynamic:

```
x01 aligned= mem128[input_1]; input_1 += 16
```

The load/store instruction does not allow an offset from the index register but does allow subsequent increment of the index register by the load amount or by another register. There are separate instructions for an unaligned 128-bit or unaligned 64-bit load or store, for an unaligned 64-bit load or store with an offset, and various other possibilities, each consuming the load/store unit for at least two cycles.

NEON includes a few permutation instructions that consume the load/store unit for one cycle: for example,

```
r = s[1] t[2] r[2,3]
```

takes a single cycle to replace `r[0]` and `r[1]` with `s[1]` and `t[2]` respectively, leaving `r[2]` and `r[3]` unchanged. This instruction reads `s` and `t` in stage N1 and writes `r` in stage N3. There are more permutation instructions that consume the load/store unit for two cycles.

Each NEON cycle dispatches at best one instruction to the arithmetic unit and one instruction to the load/store unit. These two dispatches can occur in either order. For example, a sequence of 6 single-cycle instructions of the form A LS A LS A LS will take 3 NEON cycles (A LS, A LS, A LS); a sequence LS A A LS LS A will take 3 NEON cycles (LS A, A LS, LS A); but a sequence LS LS LS A A A will take 5 NEON cycles (LS, LS, LS A, A, A).

A c-cycle instruction is dispatched in the same way as c adjacent single-cycle instructions. For example, the permutation instruction in

```
4x a2 = diag3 + diag2
   diag3 = diag3[3] diag3[0,1,2]
4x next_a2 = next_diag3 + next_diag2
```

takes two LS cycles, so overall this sequence takes two cycles (A LS, LS A). Occasional permutations thus do not cost any cycles. As another example, one can interleave two-cycle permutations with two-cycle multiplications.

3 Encrypt using a shared secret key: 5.60 cycles/byte for Salsa20

This section explains how to encrypt data with the Salsa20 stream cipher [9] at 5.60 Cortex A8 cycles/byte: e.g., 1.14 Gbps on an 800MHz core. The inner loop uses 4.58 cycles/byte and scales linearly with the number of cipher rounds; for example, Salsa20/12 uses 2.75 cycles/byte for the inner loop and 3.77 cycles/byte for the entire cipher. (These are long-message figures, but the per-message overhead is reasonably small: for example, a 1536-byte message with full Salsa20 uses 5.75 cycles/byte.)

For comparison, [29] reports that a new AES-128-CTR assembly-language implementation, contributed to OpenSSL by Polyakov, runs at 25.4 Cortex A8 cycles per byte (0.25 Gbps at 800MHz). There is no indication that this speed includes protection against software side-channel attacks; in fact, the recent paper [47] by Weiß, Heinz, and Stumpf demonstrated Cortex A8 cache-timing leakage of at least half the AES key bits from OpenSSL and several other AES implementations. An AES implementation along the lines of [23] would be protected, but the NEON cycles per byte would be far above the 7.59 Core 2 cycles per byte reported in [23].

The eBASC stream-cipher benchmarks [12] report, for Cortex A8, two other ciphers providing comparable long-message speeds: 5.77 cycles/byte for NLS v2 and 7.18 cycles/byte for TPy. NLS v2 is certainly fast, but it is limited to a 128-bit key and 2^{64} bits of output, it relies on S-box lookups that would incur extra cost to protect against cache-timing attacks, and in general it does not appear to have as large a security margin as Salsa20. We see our results as showing that the same speeds can be achieved with higher security. TPy is less competitive: it relies on random access to a large *secret* array, requiring an expensive setup for each nonce (not visible in the long-message timings) and incurring vastly higher costs for protection against cache-timing attacks.

Review of Salsa20; non-NEON bottlenecks. Salsa20 expands a 256-bit key and a 64-bit nonce into a long output stream, and xors this stream with the plaintext to produce ciphertext. The stream is generated in 64-byte blocks. The main bottleneck in generating each block is a series of 20 rounds, each consisting of 16 32-bit add-rotate-xor sequences such as the following:

```
s4 = x0 + x12
x4 ^= (s4 >>> 25)
```

This might already seem to be a perfect fit for the basic 32-bit ARM instruction set, without help from NEON. The Cortex A8 has two 32-bit execution units; addition occupies one unit for one cycle, and rotate-xor occupies one unit for one cycle. One would thus expect 320 add-rotate-xor sequences to occupy both integer execution units for 320 cycles, i.e., 5 cycles per byte.

However, there is a latency of 2 cycles between the two instructions shown above, and an overall latency of 3 cycles between the availability of `x0` and the availability of `x4`. Furthermore, the ARM architecture provides only 14 registers, but Salsa20 needs at least 17 active values: `x0` through `x15` together with a sum such as `s4`. (One can overwrite `x0` with `s4`, but only at the expense of extra arithmetic to restore `x0` afterwards.) Loads and stores occupy the execution units, taking time away from arithmetic operations. (ARM can merge two loads of adjacent registers into a single instruction, but this instruction consumes both execution units for one cycle and the first execution unit for another cycle.) There are also various overheads outside the 20-round inner loop. Compiling several different C implementations of Salsa20 with many different compiler options did not beat 15 cycles per byte.

Internal parallelization; vectorization; NEON bottlenecks. Each Salsa20 round has 4-way parallelism, with 4 independent add-rotate-xor sequences to carry out at each moment. Two parallel computations hide some latencies but require 8 loads and stores per round with our best instruction schedule; three or four parallel computations would hide all latencies but would require even more loads and stores per round.

NEON has far more space in registers, and its 128-bit arithmetic unit can perform 4 32-bit operations in each cycle. The 4 operations to carry out at each moment in Salsa20 naturally form a 4-way vector operation, at the cost of three 128-bit permutations per round. Salsa20 thus seems to be a natural fit for NEON.

However, NEON rotation consumes 3 operations as discussed in Section 2, so add-rotate-xor consumes 5 operations, at least 1.25 cycles; 5 add-rotate-xor operations per output byte consume at least 6.25 cycles per byte. Furthermore, NEON latencies are even higher than basic ARM latencies. The lowest-latency sequence of instructions for add-rotate-xor is

```
4x a0 = diag1 + diag0
4x b0 = a0 << 7
4x a0 unsigned>>= 25
   diag3 ^= b0
   diag3 ^= a0
```

with total latency 9 to the next addition: the individual latencies are 3 (N4 addition output `a0` to N1 shift input), 0 (but carried out the next cycle since the arithmetic unit is busy), 2 (N4 shift output `b0` to N2 xor input), 2 (N4 xor output `diag3` to N2 xor input), and 2 (N4 xor output `diag3` to N2 addition input). A straightforward NEON implementation cannot do better than 11.25 cycles per byte.

External parallelization. We do better by taking advantage of another level of parallelizability in Salsa20: Salsa20, like AES-CTR, generates output blocks independently as functions of a simple counter. Computing two output blocks in parallel with the following pattern of add-rotate-xor operations—

```
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
+       <<    >>     ^     ^     +       <<    >>     ^     ^
   +       <<    >>     ^     ^     +       <<    >>     ^     ^
```

—hides almost all NEON latencies, reducing our inner loop to 44 cycles per round for both blocks, i.e., 880 cycles for 20 rounds producing 128 bytes, i.e., 6.875 cycles per byte. Computing three output blocks in parallel still fits into NEON registers (with a slightly trickier pattern of operations—the most obvious patterns would need 18 registers), further reducing our inner loop to 6.25 cycles per byte, and alleviates latency issues enough to allow two-instruction rotations, but as far as we can tell this is outweighed by somewhat lower effectiveness of the speedup discussed in the next subsection.

Previous work on Salsa20 for other 128-bit vector architectures had vectorized *across* four output blocks. However, this needs at least 17 active vectors (and more to hide latencies), requiring extra instructions for loads and stores, more than the number of permutation instructions saved. This would also add overhead outside the inner loop and would interfere with the speedup described in the next subsection.

Interleaving ARM with NEON. We do better than 6.25 cycles per byte by using the basic ARM execution units to generate one block while NEON generates two blocks. Each round involves 23 NEON instructions for one block (20 instructions for four add-rotate-xor sequences, plus 3 permutation instructions), 23 NEON instructions for a second block, and 40 ARM instructions for a third block. The extra ARM instructions reduce the inner loop to $(2/3)6.875 \approx 4.58$ cycles per byte: the cycles for the loop are exactly the same but the loop produces $1.5\times$ as much output.

We are pushing this technique extremely close to an important Cortex A8 limit. The limit is that the entire core decodes at most two instructions per cycle, whether the instructions are ARM instructions or NEON instructions. The 880 cycles that we spend for 128 NEON output bytes have 1760 instruction slots, while we use only 920 NEON instructions, leaving 840 free slots; we use 800 of these slots for ARM instructions that generate 64 additional output bytes, and an additional 35 slots for loop control to avoid excessive code size. (Register pressure forced us to spill the loop counter, and each branch instruction has a hidden cost of 3 slots; we ended up unrolling 4 rounds.) Putting even marginally more work on the ARM unit would slow down the NEON processing, and an easy quantitative analysis shows that this would slow down the cipher as a whole.

The same limit makes ARM instructions far less effective for, e.g., the computations modulo $2^{255}-19$ discussed later in this paper. These computations are large enough that they require many NEON loads and stores alongside arithmetic, often consuming both of the instruction slots available in a cycle. There are still some slots for ARM instructions, but these computations require an even larger number of ARM loads and stores, leaving very few slots for ARM arithmetic instructions. Furthermore, these computations are dominated by multiplications rather than rotations, and even full-speed ARM multiplications have only a fraction of the power of NEON multiplications.

Minimizing overhead. The above discussion concentrates on the performance of the Salsa20 inner loop, but there are also overheads for initializing and finalizing each block, reading plaintext, and generating ciphertext.

The 64-byte Salsa20 output block consists of four vectors `x0 x1 x2 x3`, `x4 x5 x6 x7`, `x8 x9 x10 x11`, and `x12 x13 x14 x15` that must be xor'ed with plaintext to produce ciphertext. NEON uses 0.125 cycles/byte to read potentially unaligned plaintext, and 0.125 cycles/byte to write potentially unaligned ciphertext, for an overhead of 0.25 cycles/byte; ARM is slower. It should be possible to reduce this overhead, at some cost in code size, by overlapping memory access with computation, but we have not yet done this.

The Salsa20 inner loop naturally uses and produces "diagonal" vectors `x0 x5 x10 x15`, `x4 x9 x13 x3`, etc. Converting these diagonal vectors to the output vectors `x0 x1 x2 x3` etc. poses an interesting challenge for NEON's permutation instructions. We use the following short sequence of instructions (and gratefully acknowledge optimization assistance from Tanja Lange):

```
r0 = ...                          # x0 x5 x10 x15
r4 = ...                          # x4 x9 x14 x3
r12 = ...                         # x12 x1 x6 x11
r8 = ...                          # x8 x13 x2 x7
t4 = r0[1] r4[0] t4[2,3]          # x5 x4 - -
```

```
t12 = t12[0,1] r0[3] r4[2]       # - - x15 x14
r0 = (abab & r0) | (~abab & r12) # x0 x1 x10 x11
t4 = t4[0,1] r8[3] r12[2]        # x5 x4 x7 x6
t12 = r8[1] r12[0] t12[2,3]      # x13 x12 x15 x14
r8 = (abab & r8) | (~abab & r4)  # x8 x9 x2 x3
r4 = t4[1]t4[0]t4[3]t4[2]        # x4 x5 x6 x7
r12 = t12[1]t12[0]t12[3]t12[2]   # x12 x13 x14 x15
r0 r8 = r0[0] r8[1] r8[0] r0[1]  # x0 x1 x2 x3 x8 x9 x10 x11
```

There are 7 single-cycle permutations here, consuming 0.11 cycles/byte, and 2 two-cycle arithmetic instructions (using `abab`) interleaved with the permutations. Similar comments apply to block initialization. These and other overheads increase the overall encryption costs to 5.60 cycles/byte.

4 Authenticate using a shared secret key: 2.30 cycles/byte for Poly1305

This section explains how to compute the Poly1305 message-authentication code [5] at 2.30 Cortex A8 cycles/byte: e.g., 2.78 Gbps on an 800MHz core. Authenticated encryption with Salsa20 and Poly1305 takes just 7.90 cycles/byte.

For comparison, [29] reports 50 Cortex A8 cycles/byte for AES-GCM and 28.9 cycles/byte for its proposed AES-OCB3; compared to the 25.4 cycles/byte of AES-CTR encryption, authentication adds 25 or 3.5 cycles/byte respectively. GCM, OCB3, and Poly1305 guarantee that attacks are as difficult as breaking the underlying cipher, with similar quantitative security bounds. Another approach, without this guarantee, is HMAC using a hash function; the Cortex A8 speed leaders in the eBASH hash-function benchmarks [12] are MD5 at 6.04 cycles/byte, Edon-R at 9.76 cycles/byte, Shabal at 12.94 cycles/byte, BMW at 13.55 cycles/byte, and Skein at 15.26 cycles/byte.

One of these authentication speeds, the "free" 3.5-cycle/byte authentication in OCB3, is within a factor of 2 of our Poly1305 speed. However, OCB3 also has two important disadvantages. First, OCB3 cannot be combined with a fast stream cipher such as Salsa20—it requires a block cipher, as discussed in [29]. Second, rejecting an OCB3 forgery requires taking the time to decrypt the forgery, a full 28.9 cycles/byte; Poly1305 rejects forgeries an order of magnitude more quickly.

Review of Poly1305. Poly1305 reads a *one-time* 32-byte secret key and a message of any length. It chops the message into 128-bit little-endian integers (and a final b-bit integer with $b \leq 128$), adds 2^{128} to each integer (and 2^b to the final integer) to obtain components $m[0], m[1], \ldots, m[\ell-1]$, and produces the 16-byte authenticator

$$(((m[0]r^\ell + m[1]r^{\ell-1} + \cdots + m[\ell-1]r) \bmod 2^{130} - 5) + s) \bmod 2^{128}$$

where r and s are components of the secret key. "One time" has the same meaning as for a one-time pad: each message has a new key. If these one-time keys are truly random then the attacker is reduced to blind guessing; see [5] for quantitative bounds on the attacker's forgery chance. If these keys are instead produced as cipher outputs from a long-term key then security relies on the presumed difficulty of distinguishing the cipher outputs from random.

Readers familiar with the GCM authenticated-encryption mode [32] will recognize that Poly1305 shares the polynomial-evaluation structure of the GMAC authenticator inside GCM. The general structure was introduced by den Boer [18], Johansson, Kabatianskii, and Smeets [24], and independently Taylor [43]; concrete examples include [39], [34], [4], [28], and [27]. But these proposals differ in many details, notably the choice of finite field: a field of size 2^{128} for GCM, for example, and integers modulo $2^{130} - 5$ for Poly1305.

Efficient authentication in software relies primarily on fast multiplication in this field, and secondarily on fast conversion of message bytes into elements of the field. Efficient authentication under a *one-time* key (addressing the security issues discussed in [4, Section 8, Notes], [8, Sections 2.4–2.5], [21], [14], etc.) means that one cannot afford to precompute large tables of multiples of r;

we count the costs of all precomputation. Avoiding the possibility of cache-timing attacks means that one cannot use variable-index table lookups; see, e.g., the discussion of GCM security in [23, Section 2.3].

Multiplication mod $2^{130}-5$ on NEON. We represent an integer f modulo $2^{130}-5$ in radix 2^{26} as $f_0+2^{26}f_1+2^{52}f_2+2^{78}f_3+2^{104}f_4$. At the end of the computation we reduce each f_i below 2^{26}, and reduce f to the interval $\{0,1,\ldots,2^{130}-6\}$, but earlier in the computation we use standard lazy-reduction techniques, allowing wider ranges of f and of f_i.

The most attractive NEON multipliers are the paired 32-bit multipliers, which as discussed in Section 2 produce two 64-bit products every two cycles, including free additions. The product of $f_0+2^{26}f_1+\cdots$ and $g_0+2^{26}g_1+\cdots$ is $h_0+2^{26}h_1+\cdots$ modulo $2^{130}-5$ where

$$\begin{aligned}
h_0 &= f_0g_0+5f_1g_4+5f_2g_3+5f_3g_2+5f_4g_1,\\
h_1 &= f_0g_1+\ f_1g_0+5f_2g_4+5f_3g_3+5f_4g_2,\\
h_2 &= f_0g_2+\ f_1g_1+\ f_2g_0+5f_3g_4+5f_4g_3,\\
h_3 &= f_0g_3+\ f_1g_2+\ f_2g_1+\ f_3g_0+5f_4g_4,\\
h_4 &= f_0g_4+\ f_1g_3+\ f_2g_2+\ f_3g_1+\ f_4g_0,
\end{aligned}$$

all of which are smaller than $2^{64}/195$ if each f_i and g_i is bounded by 2^{26}. Evidently somewhat larger inputs f_i and g_i, products of sums of inputs, sums of several outputs, etc. do not pose any risk of 64-bit overflow. This computation (performed from right to left to absorb all sums into products) involves 25 generic multiplications and 4 multiplications by 5, but it is better to eliminate the multiplications by 5 in favor of precomputing $5g_1,5g_2,5g_3,5g_4$, in part because those are 32-bit multiplications and in part because a multiplication input is often reused.

Rather than vectorizing within a message block, and having to search for 12 convenient pairs of 32-bit multiplications in the pattern of 25 multiplications shown above, we simply vectorize across two message blocks, using a well-known parallelization of Horner's rule. For example, for $\ell=10$, we compute

$$\begin{aligned}
&((((m[0]r^2+m[2])r^2+m[4])r^2+m[6])r^2+m[8])r^2\\
&+((((m[1]r^2+m[3])r^2+m[5])r^2+m[7])r^2+m[9])r
\end{aligned}$$

by starting with the vector $(m[0],m[1])$, multiplying by the vector (r^2,r^2), adding $(m[2],m[3])$, multiplying by (r^2,r^2), etc. The integer $m[0]$ is actually represented as five 32-bit words, so the vector $(m[0],m[1])$ is actually represented as five vectors of 32-bit words. The 25 multiplications shown above, times two blocks, then trivially use 25 NEON multiplication instructions costing 50 cycles, i.e., 1.5625 cycles per byte. There are, however, also overheads for reading the message and reducing the product, as discussed below.

Reduction. The product obtained above can be safely added to a new message block but must be reduced before it can be used as input to another multiplication. To reduce a large coefficient h_0, we carry $h_0\to h_1$; this means replacing (h_0,h_1) with $(h_0 \bmod 2^{26}, h_1+\lfloor h_0/2^{26}\rfloor)$. Similar comments apply to the other coefficients. Carrying $h_4\to h_0$ means replacing (h_4,h_0) with $(h_4 \bmod 2^{26}, h_0+5\lfloor h_4/2^{26}\rfloor)$, again taking advantage of the sparsity of $2^{130}-5$.

NEON uses 1 cycle for a pair of 64-bit shifts, 1 cycle for a pair of 64-bit masks, and 1 cycle for a pair of 64-bit additions, for a total of 3 cycles for a pair of carries (plus 2 cycles for $h_4\to h_0$). A chain of six carries $h_0\to h_1\to h_2\to h_3\to h_4\to h_0\to h_1$ is adequate for subsequent multiplications: it leaves h_1 below $2^{26}+2^{13}$ and each other h_i below 2^{26}. However, each step in this chain has latency at least 5, and even aggressive interleaving of carries into the computations of h_i would eliminate only a few of the resulting idle cycles. We instead carry $h_0\to h_1$ and $h_3\to h_4$, then $h_1\to h_2$ and $h_4\to h_0$, then $h_2\to h_3$ and $h_0\to h_1$, then $h_3\to h_4$, spending 3 cycles to eliminate latency problems. The selection of initial indices $(0,3)$ here allows the longer carry $h_4\to h_0$ to overlap two independent carries $h_1\to h_2\to h_3$; we actually interleave $h_0\to h_1\to h_2\to h_3\to h_4$ with $h_3\to h_4\to h_0\to h_1$, being careful to keep the separate uses of h_i away from each other.

This approach consumes 23 cycles for two blocks, i.e., 0.71875 cycles per byte. As message lengths grow it becomes better to retreat from Horner's method, for example computing

$$((m[0]r^4 + m[2]r^2 + m[4])r^4 + m[6]r^2 + m[8])r^2$$
$$+ ((m[1]r^4 + m[3]r^2 + m[5])r^4 + m[7]r^2 + m[9])r$$

by starting with $(m[0], m[1])$ and $(m[2], m[3])$, multiplying by (r^4, r^4) and (r^2, r^2) respectively, adding, adding $(m[4], m[5])$, *then* reducing, etc. This eliminates half of the reductions at the expense of extending the precomputation from (r^2, r^2) to (r^4, r^4). One can easily eliminate more reductions with more precomputation, but one pays for precomputation linearly in both time and space, while the benefit becomes smaller and smaller.

For comparison, [4, Section 6] precomputed 97 powers of r for a polynomial evaluation in another field. The number 97 was chosen to just barely avoid overflow of sums of 97 intermediate values; [4] did not count the cost of precomputation. Of course, when we report long-message performance figures we blind ourselves to any constant amount of precomputation, but beyond those figures we are also careful to avoid excessive precomputation (and, for similar reasons, excessive code size). We thus settled on eliminating half of the reductions.

Reading the message. The inner loop in our computation, with half reductions as described above, computes $fr^4 + m[i]r^2 + m[i+2]$. One input is an accumulator f; the output is written on top of f for the next pass through the loop. Two more inputs are r^2 and r^4, both precomputed. The last two inputs are message blocks $m[i]$ and $m[i+2]$; the inner loop loads these blocks and converts them to radix 2^{26}. The following paragraphs discuss the costs of this conversion.

The same computations are carried out in parallel on $m[i+1]$ and $m[i+3]$, using another accumulator. We suppress further mention of this straightforward vectorization: for example, when we say below that NEON takes 0.5 cycles for a 64-bit shift involved in $m[i]$, what we actually mean is that NEON takes 1 cycle for a pair of 64-bit shifts, where the first shift is used for $m[i]$ and the second is used for $m[i+1]$.

Loading $m[i]$ produces a vector (m_0, m_1, m_2, m_3) representing the integer $m_0 + 2^{32}m_1 + 2^{64}m_2 + 2^{96}m_3$. Our goal here is to represent the same integer (plus 2^{128}) in radix 2^{26} as $c_0 + 2^{26}c_1 + 2^{52}c_2 + 2^{78}c_3 + 2^{104}c_4$. A shift of the 64 bits (m_2, m_3) down by 40 bits produces exactly c_4. A shift of (m_2, m_3) down by 14 bits does not produce exactly c_3, and a shift of (m_1, m_2) down by 20 bits does not produce exactly c_2, but a single 64-bit mask then produces (c_2, c_3). Similar comments apply to (c_0, c_1), except that c_0 does not require a shift.

Overall there are seven 64-bit arithmetic instructions here (four shifts, two masks, and one addition to c_4 to handle the 2^{128}), consuming 3.5 cycles for each 16-byte block. There is also a two-cycle (potentially unaligned) load, along with just six single-cycle permutation instructions; NEON has an arithmetic instruction that combines a 64-bit right shift (by up to 32 bits) with an extraction of the bottom 32 bits of the result, eliminating some 64-bit-to-32-bit shuffling.

The second message block $m[i+2]$ has a different role in $fr^4 + m[i]r^2 + m[i+2]$: it is added to the output rather than the input. We take advantage of this by loading $m[i+2]$ into a vector (m_0, m_1, m_2, m_3) and adding $m_0 + 2^{32}m_1 + 2^{64}m_2 + 2^{96}m_3$ into a multiplication result $h_0 + 2^{26}h_1 + 2^{52}h_2 + 2^{78}h_3 + 2^{104}h_4$ *before* carrying the result. This means simply adding m_0 into h_0, adding 2^6m_1 into h_1, etc. We absorb the additions into multiplications by scheduling $m[i+2]$ before the computation of h. The only remaining costs for $m[i+2]$ are a few shifts such as 2^6m_1, one operation to add 2^{128}, and various permutations.

The conversion of $m[i]$ and $m[i+2]$ costs, on average, 0.171875 cycles/byte for arithmetic instructions. Our total cost for NEON arithmetic in Poly1305 is 2.09375 cycles/byte: 1.5625 cycles/byte for one multiplication per block, 0.359375 cycles/byte for half a reduction per block, and 0.171875 cycles/byte for input conversion. We have not yet managed to perfectly schedule the inner loop: right now it takes 147 cycles for 64 bytes, slightly above the 134 cycles of arithmetic, so our software computes Poly1305 at 2.30 cycles/byte.

5 Compute a shared secret key for a new public key: 527102 cycles for Curve25519; sign and verify: 368212 and 650102 cycles for Ed25519

This section explains how to compute the Curve25519 Diffie–Hellman function [6], obtaining a 32-byte shared secret from Alice's 32-byte secret key and Bob's 32-byte public key, in 527102 Cortex A8 cycles: e.g., 1517/second on an 800MHz core. This section also explains how to sign and verify messages in the Ed25519 public-key signature system [10] in, respectively, 368212 and 650102 Cortex A8 cycles: e.g., 2172/second and 1230/second on an 800MHz core. Ed25519 public keys are 32 bytes, and signatures are 64 bytes.

For comparison, `openssl speed` on the same machine reports

- 424.2 RSA-2048 verifications per second (1.9 million cycles),
- 11.1 RSA-2048 signatures per second (72 million cycles),
- 88.6 NIST P-256 Diffie–Hellman operations per second (9.0 million cycles),
- 388.8 NIST P-256 signatures per second (2.1 million cycles), and
- 74.5 NIST P-256 verifications per second (10.7 million cycles).

Morozov, Tergino, and Schaumont [33] report two speeds for "secp224r1" Diffie–Hellman: 15609 microseconds on a 500MHz Cortex A8 (7.8 million cycles), and 6043 microseconds on a 360MHz DSP (2 million DSP cycles) included in the same CPU, a TI OMAP 3530. Curve25519 and Ed25519 have a higher security level than secp224r1 and 2048-bit RSA; it is also not clear which of the previous speeds include protection against side-channel attacks.

Review of Curve25519 and Ed25519. Curve25519 and Ed25519 are elliptic-curve systems. Key generation is fixed-base-point single-scalar multiplication: Bob's public key is a multiple $B = bP$ of a standard base point P on a standard curve. Bob's secret key is the integer b.

Curve25519's Diffie–Hellman function is variable-base-point single-scalar multiplication: Alice, given Bob's public key B, computes aB where a is Alice's secret key. The secret shared by Alice and Bob is simply a hash of aB; this secret is used, for example, as a long-term key for Salsa20, which in turn is used to generate encryption pads and Poly1305 authentication keys.

Signing in Ed25519 consists primarily of fixed-base-point single-scalar multiplication. (We make the standard assumption that messages are short; hashing time is the bottleneck for very long messages. Our measurements use 59-byte messages, as in [12].) Signing is much faster than Diffie–Hellman: it exploits precomputed multiples of P in various standard ways. Verification in Ed25519 is slower than Diffie–Hellman: it consists primarily of double-scalar multiplication.

The Curve25519 elliptic curve is the Montgomery curve $y^2 = x^3 + 486662x^2 + x$ modulo $2^{255} - 19$, with a unique point of order 2. The Ed25519 elliptic curve is the twisted Edwards curve $-x^2 + y^2 = 1 - (121665/121666)x^2y^2$ modulo $2^{255} - 19$, also with a unique point of order 2. These two curves have an "efficient birational equivalence" and therefore have the same security.

Montgomery curves are well known to allow efficient variable-base-point single-scalar multiplication. Edwards curves are well known to allow a wider variety of efficient elliptic-curve operations, including double-scalar multiplication. These fast scalar-multiplication methods are "complete": they are sequences of additions, multiplications, etc. that always produce the right answer, with no need for comparisons, branches, etc. Completeness was proven by Bernstein [6] for single-scalar multiplication on any Montgomery curve having a unique point of order 2, and by Bernstein and Lange [11] for arbitrary group operations on any Edwards curve having a unique point of order 2.

The main loop in Curve25519, executed 255 times, has four additions of integers modulo $2^{255} - 19$, four subtractions, two conditional swaps (which must be computed with arithmetic rather than branches or variable array lookups), four squarings, one multiplication by the constant 121666, and five generic multiplications. There is also a smaller final loop (a field inversion), consisting of 254 squarings and 11 multiplications. Similar comments apply to Ed25519 signing and Ed25519 verification.

Multiplication mod $2^{255}-19$ on NEON. We use radix $2^{25.5}$, imitating the floating-point representation in [6, Section 4] but with unscaled integers rather than scaled floating-point numbers: we represent an integer f modulo $2^{255}-19$ as

$$f_0+2^{26}f_1+2^{51}f_2+2^{77}f_3+2^{102}f_4+2^{128}f_5+2^{153}f_6+2^{179}f_7+2^{204}f_8+2^{230}f_9$$

where, as in Section 4, the allowable ranges of f_i vary through the computation.

We use signed integers f_i rather than unsigned integers: for example, when we carry $f_0 \to f_1$ we reduce f_0 to the range $[-2^{25}, 2^{25}]$ rather than $[0, 2^{26}]$. This complicates carries, replacing a mask with a shift and subtraction, but saves one bit in products of reduced coefficients, allowing us to safely compute various products of sums without carrying the sums. This was unnecessary in the previous section, in part because the 5 in $2^{130}-5$ is smaller than the 19 in $2^{255}-19$, in part because 130 is smaller than 255, and in part because the sums of inputs and outputs naturally appearing in the previous section have fewer terms than the sums that appear in these elliptic-curve computations.

The product of $f_0+2^{26}f_1+2^{51}f_2+\cdots$ and $g_0+2^{26}g_1+2^{51}g_2+\cdots$ is $h_0+2^{26}h_1+2^{51}h_2+\cdots$ modulo $2^{255}-19$ where

$$\begin{aligned}
h_0 &= f_0g_0+ 38f_1g_9+ 19f_2g_8+ 38f_3g_7+ 19f_4g_6+ 38f_5g_5+ 19f_6g_4+ 38f_7g_3+ 19f_8g_2+ 38f_9g_1\\
h_1 &= f_0g_1+ f_1g_0+ 19f_2g_9+ 19f_3g_8+ 19f_4g_7+ 19f_5g_6+ 19f_6g_5+ 19f_7g_4+ 19f_8g_3+ 19f_9g_2\\
h_2 &= f_0g_2+ 2f_1g_1+ f_2g_0+ 38f_3g_9+ 19f_4g_8+ 38f_5g_7+ 19f_6g_6+ 38f_7g_5+ 19f_8g_4+ 38f_9g_3\\
h_3 &= f_0g_3+ f_1g_2+ f_2g_1+ f_3g_0+ 19f_4g_9+ 19f_5g_8+ 19f_6g_7+ 19f_7g_6+ 19f_8g_5+ 19f_9g_4\\
h_4 &= f_0g_4+ 2f_1g_3+ f_2g_2+ 2f_3g_1+ f_4g_0+ 38f_5g_9+ 19f_6g_8+ 38f_7g_7+ 19f_8g_6+ 38f_9g_5\\
h_5 &= f_0g_5+ f_1g_4+ f_2g_3+ f_3g_2+ f_4g_1+ f_5g_0+ 19f_6g_9+ 19f_7g_8+ 19f_8g_7+ 19f_9g_6\\
h_6 &= f_0g_6+ 2f_1g_5+ f_2g_4+ 2f_3g_3+ f_4g_2+ 2f_5g_1+ f_6g_0+ 38f_7g_9+ 19f_8g_8+ 38f_9g_7\\
h_7 &= f_0g_7+ f_1g_6+ f_2g_5+ f_3g_4+ f_4g_3+ f_5g_2+ f_6g_1+ f_7g_0+ 19f_8g_9+ 19f_9g_8\\
h_8 &= f_0g_8+ 2f_1g_7+ f_2g_6+ 2f_3g_5+ f_4g_4+ 2f_5g_3+ f_6g_2+ 2f_7g_1+ f_8g_0+ 38f_9g_9\\
h_9 &= f_0g_9+ f_1g_8+ f_2g_7+ f_3g_6+ f_4g_5+ f_5g_4+ f_6g_3+ f_7g_2+ f_8g_1+ f_9g_0.
\end{aligned}$$

The extra factors of 2 appear because $2^{25.5}$ is not an integer. We precompute $2f_1, 2f_3, 2f_5, 2f_7, 2f_9$ and $19g_1, 19g_2, \ldots, 19g_9$; each h_i is then a sum of ten products of precomputed quantities.

Most multiplications appear as independent pairs, computing fg and $f'g'$ in parallel, in the elliptic-curve formulas we use. We vectorize across these multiplications: we start from 20 64-bit vectors such as (f_0, f_0') and (g_0, g_0'), precompute 14 64-bit vectors such as $(2f_1, 2f_1')$ and $(19g_1, 19g_1')$, and then accumulate 10 128-bit vectors such as (h_0, h_0'). By scheduling operations carefully we fit these 54 64-bit quantities into the 32 available 64-bit registers with a moderate number of loads and stores.

Some multiplications do not appear as pairs. For those cases we vectorize *within* one multiplication by the following strategy. Accumulate the vectors $(f_0g_0, 2f_1g_1)$ and $(19f_2g_8, 38f_3g_9)$ and $(19f_4g_6, 38f_5g_7)$ and $(19f_6g_4, 38f_7g_5)$ and $(19f_8g_2, 38f_9g_3)$ into (h_0, h_2); accumulate $(f_0g_2, 2f_1g_3)$ etc. into (h_2, h_4); and so on through (h_8, h_0). Also accumulate $(f_1g_2, 19f_8g_3)$, (f_3g_0, f_0g_1), etc. into (h_3, h_1); accumulate $(f_1g_4, 19f_8g_5)$ etc. into (h_5, h_3); and so on through (h_1, h_9). Each vector added here is a product of two of the following 27 precomputed vectors:

- $(f_0, 2f_1), (f_2, 2f_3), (f_4, 2f_5), (f_6, 2f_7), (f_8, 2f_9)$;
- $(f_1, f_8), (f_3, f_0), (f_5, f_2), (f_7, f_4), (f_9, f_6)$;
- $(g_0, g_1), (g_2, g_3), (g_4, g_5), (g_6, g_7)$;
- $(g_0, 19g_1), (g_2, 19g_3), (g_4, 19g_5), (g_6, 19g_7), (g_8, 19g_9)$;
- $(19g_2, 19g_3), (19g_4, 19g_5), (19g_6, 19g_7), (19g_8, 19g_9)$;
- $(19g_2, g_3), (19g_4, g_5), (19g_6, g_7), (19g_8, g_9)$.

We tried several other strategies, pairing inputs and outputs in various ways, before settling on this strategy. All of the other strategies used more precomputed vectors, requiring more loads and stores.

Reduction, squaring, etc. Reduction follows an analogous strategy to Section 4. One complication is that each carry has an extra operation, as mentioned above. Another complication for vectorizing a single multiplication is that the shift distances are sometimes 26 bits and sometimes 25 bits; we vectorize carrying $(h_0, h_4) \to (h_1, h_5)$, for example, but would not have been able to vectorize carrying $(h_0, h_5) \to (h_1, h_6)$.

For squaring, like multiplication, we vectorize across two independent operations when possible, and otherwise vectorize within one operation. Squarings are serialized in square-root computations (for decompressing short signatures) and in inversions (for converting scalar-multiplication results to affine coordinates), but the critical bottlenecks are elliptic-curve operations, and squarings come in convenient pairs in all of the elliptic-curve formulas that we use.

In the end arithmetic consumes 150 cycles in generic multiplication (called 1286 times in Curve25519), 105 cycles in squaring (called 1274 times), 67 cycles in multiplication by 121666 (called 255 times), 3 cycles in addition (called 1020 times), 3 cycles in subtraction (called 1020 times), and 12 cycles in conditional swaps (called 512 times), explaining fewer than 400000 cycles. The most important source of overhead in our current Curve25519 performance, 527102 cycles, is non-arithmetic instructions at the beginning and end of each function. We are working on addressing this by inlining all functions into the main loop and scheduling the main loop as a whole, and we anticipate then coming much closer to the lower bound, as in Salsa20 and Poly1305.

Similar comments apply to Ed25519. Many Ed25519 cycles (about 50000 cycles in signing and 25000 in verification) are consumed by the SHA-512 implementation selected by SUPERCOP [12]. We see ample room for improving the SHA-512 implementation on the Cortex A8 but have not bothered doing so: the Ed25519 paper [10] recommends switching to Ed25519-SHA-3.

References

[1] — (no editor), *9th IEEE symposium on application specific processors*, Institute of Electrical and Electronics Engineers, 2011. See [33].

[2] Onur Aciiçmez, Billy Bob Brumley, Philipp Grabher, *New results on instruction cache attacks*, in CHES 2010 [31] (2010), 110–124. Citations in this document: §1.

[3] ARM Limited, *Cortex-A8 technical reference manual, revision r3p2*, 2010. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html. Citations in this document: §2.

[4] Daniel J. Bernstein, *Floating-point arithmetic and message authentication* (1999). URL: http://cr.yp.to/papers.html#hash127. Citations in this document: §4, §4, §4, §4.

[5] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in FSE 2005 [20] (2005), 32–49. URL: http://cr.yp.to/papers.html#poly1305. Citations in this document: §1, §4, §4.

[6] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [48] (2006), 207–228. URL: http://cr.yp.to/papers.html#curve25519. Citations in this document: §1, §5, §5, §5.

[7] Daniel J. Bernstein, `qhasm` *software package* (2007). URL: http://cr.yp.to/qhasm.html. Citations in this document: §2.

[8] Daniel J. Bernstein, *Polynomial evaluation and message authentication* (2007). URL: http://cr.yp.to/papers.html#pema. Citations in this document: §4.

[9] Daniel J. Bernstein, *The Salsa20 family of stream ciphers*, in [37] (2008), 84–97. URL: http://cr.yp.to/papers.html#salsafamily. Citations in this document: §1, §3.

[10] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 [36] (2011). URL: http://eprint.iacr.org/2011/368. Citations in this document: §1, §5, §5.

[11] Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [30] (2007), 29–50. URL: http://eprint.iacr.org/2007/286. Citations in this document: §5.

[12] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 5 March 2012 (2012). URL: http://bench.cr.yp.to. Citations in this document: §1, §3, §4, §5, §5.

[13] Daniel J. Bernstein, Tanja Lange, Peter Schwabe, *The security impact of a new cryptographic library* (2011). URL: http://eprint.iacr.org/2011/646. Citations in this document: §1.

[14] John Black, Martin Cochran, *MAC reforgeability*, in FSE 2009 [19] (2009), 345–362. URL: http://eprint.iacr.org/2006/095. Citations in this document: §4.

[15] Anne Canteaut, Kapalee Viswanathan (editors), *Progress in cryptology—INDOCRYPT 2004, 5th international conference on cryptology in India, Chennai, India, December 20–22, 2004, proceedings*, Lecture Notes in Computer Science, 3348, Springer, 2004. ISBN 3-540-24130-2. See [32].

[16] Christophe Clavier, Kris Gaj (editors), *Cryptographic hardware and embedded systems—CHES 2009, 11th international workshop, Lausanne, Switzerland, September 6–9, 2009, proceedings*, Lecture Notes in Computer Science, 5747, Springer, 2009. ISBN 978-3-642-04137-2. See [23].

[17] Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in Africacrypt 2009 [**35**] (2009), 368–385. URL: http://cryptojedi.org/users/peter/#celldh. Citations in this document: §1.
[18] Bert den Boer, *A simple and key-economical unconditional authentication scheme*, Journal of Computer Security **2** (1993), 65–71. ISSN 0926–227X. Citations in this document: §4.
[19] Orr Dunkelman (editor), *Fast software encryption, 16th international workshop, FSE 2009, Leuven, Belgium, February 22–25, 2009, revised selected papers*, Lecture Notes in Computer Science, 5665, Springer, 2009. ISBN 978-3-642-03316-2. See [14].
[20] Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3-540-26541-4. See [5].
[21] Helena Handschuh, Bart Preneel, *Key-recovery attacks on universal hash function based MAC algorithms*, in CRYPTO 2008 [**46**] (2008), 144–161. Citations in this document: §4.
[22] Tor Helleseth (editor), *Advances in cryptology—EUROCRYPT '93, workshop on the theory and application of cryptographic techniques, Lofthus, Norway, May 23–27, 1993, proceedings*, Lecture Notes in Computer Science, 765, Springer, 1994. ISBN 3-540-57600-2. See [24].
[23] Emilia Käsper, Peter Schwabe, *Faster and timing-attack resistant AES-GCM*, in CHES 2009 [**16**] (2009), 1–17. URL: http://eprint.iacr.org/2009/129. Citations in this document: §3, §3, §4.
[24] Thomas Johansson, Gregory Kabatianskii, Ben J. M. Smeets, *On the relation between A-codes and codes correcting independent errors*, in EUROCRYPT '93 [**22**] (1994), 1–11. Citations in this document: §4.
[25] Antoine Joux (editor), *Fast software encryption—18th international workshop, FSE 2011, Lyngby, Denmark, February 13–16, 2011, revised selected papers*, Lecture Notes in Computer Science, 6733, Springer, 2011. ISBN 978-3-642-21701-2. See [29].
[26] Neal Koblitz (editor), *Advances in cryptology—CRYPTO '96*, Lecture Notes in Computer Science, 1109, Springer, 1996. See [39].
[27] Tadayoshi Kohno, John Viega, Doug Whiting, *CWC: a high-performance conventional authenticated encryption mode*, in FSE 2004 [**38**] (2004), 408–426. Citations in this document: §4.
[28] Ted Krovetz, Phillip Rogaway, *Fast universal hashing with small keys and no preprocessing: the PolyR construction* (2000). URL: http://www.cs.ucdavis.edu/~rogaway/papers/poly.htm. Citations in this document: §4.
[29] Ted Krovetz, Philip Rogaway, *The software performance of authenticated-encryption modes*, in FSE 2011 [**25**] (2011), 306–327. URL: http://www.cs.ucdavis.edu/~rogaway/papers/ae.pdf. Citations in this document: §3, §4, §4.
[30] Kaoru Kurosawa (editor), *Advances in cryptology—ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, proceedings*, Lecture Notes in Computer Science, 4833, Springer, 2007. ISBN 978-3-540-76899-9. See [11].
[31] Stefan Mangard, François-Xavier Standaert (editors), *Cryptographic hardware and embedded systems, CHES 2010, 12th international workshop, Santa Barbara, CA, USA, August 17–20, 2010, proceedings*, Lecture Notes in Computer Science, 6225, Springer, 2010. ISBN 978-3-642-15030-2. See [2].
[32] David A. McGrew, John Viega, *The security and performance of the Galois/Counter mode (GCM) of operation*, in INDOCRYPT 2004 [**15**] (2004), 343–355. URL: http://eprint.iacr.org/2004/193. Citations in this document: §4.
[33] Sergey Morozov, Christian Tergino, Patrick Schaumont, *System integration of elliptic curve cryptography on an OMAP Platform*, in [**1**] (2011), 52–57. URL: http://rijndael.ece.vt.edu/schaum/papers/2011sasp.pdf. Citations in this document: §5.
[34] Wim Nevelsteen, Bart Preneel, *Software performance of universal hash functions*, in EUROCRYPT '99 [**41**] (1999), 24–41. Citations in this document: §4.
[35] Bart Preneel (editor), *Progress in cryptology—AFRICACRYPT 2009, second international conference on cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009, proceedings*, Lecture Notes in Computer Science, 5580, Springer, 2009. See [17].
[36] Bart Preneel, Tsuyoshi Takagi (editors), *Cryptographic hardware and embedded systems—CHES 2011, 13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings*, Lecture Notes in Computer Science, Springer, 2011. ISBN 978-3-642-23950-2. See [10].
[37] Matthew Robshaw, Olivier Billet (editors), *New stream cipher designs*, Lecture Notes in Computer Science, 4986, Springer, 2008. ISBN 978-3-540-68350-6. See [9].
[38] Bimal K. Roy, Willi Meier (editors), *Fast software encryption, 11th international workshop, FSE 2004, Delhi, India, February 5–7, 2004, revised papers*, Lecture Notes in Computer Science, 3017, Springer, 2004. ISBN 3-540-22171-9. See [27].
[39] Victor Shoup, *On fast and provably secure message authentication based on universal hashing*, in CRYPTO '96 [**26**] (1996), 313–328. URL: http://www.shoup.net/papers. Citations in this document: §4.
[40] Étienne Sobole, *Calculateur de cycle pour le Cortex A8* (2012). URL: http://pulsar.webshaker.net/ccc/index.php. Citations in this document: §2.
[41] Jacques Stern (editor), *Advances in cryptology—EUROCRYPT '99*, Lecture Notes in Computer Science, 1592, Springer, 1999. ISBN 3-540-65889-0. MR 2000i:94001. See [34].
[42] Douglas R. Stinson (editor), *Advances in cryptology—CRYPTO '93: 13th annual international cryptology conference, Santa Barbara, California, USA, August 22–26, 1993, proceedings*, Lecture Notes in Computer Science, 773, Springer, 1994. ISBN 3-540-57766-1, 0-387-57766-1. See [43].

[43] Richard Taylor, *An integrity check value algorithm for stream ciphers*, in CRYPTO '93 [**42**] (1994), 40–48. Citations in this document: §4.

[44] Eran Tromer, Dag Arne Osvik, Adi Shamir, *Efficient cache attacks on AES, and countermeasures*, Journal of Cryptology **23** (2010), 37–71. URL: http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf. Citations in this document: §1.

[45] David Ulevitch, *DNSCrypt—critical, fundamental, and about time* (2011). URL: http://blog.opendns.com/2011/12/06/dnscrypt-%E2%80%93-critical-fundamental-and-about-time/. Citations in this document: §1.

[46] David Wagner (editor), *Advances in cryptology—CRYPTO 2008, 28th annual international cryptology conference, Santa Barbara, CA, USA, August 17–21, 2008, proceedings*, Lecture Notes in Computer Science, 5157, Springer, 2008. ISBN 978-3-540-85173-8. See [21].

[47] Michael Weiß, Benedikt Heinz, Frederic Stumpf, *A cache timing attack on AES in virtualization environments*, Proceedings of Financial Cryptography 2012, to appear (2012). Citations in this document: §1, §3.

[48] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography—9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [6].