Constructions based on the AES round and polynomial multiplications that are efficient on modern processor architectures

Shay Gueron
University of Haifa, Israel and Meta, USA
sgueron@univ.haifa.ac.il

Abstract. The Advanced Encryption Standard (AES) has become the most frequently used block cipher since standardization in 2001. Processor instructions that speed up AES computations and polynomial multiplication in GF (2^n) were introduced in 2009 and have become part of almost all 64-bit modern processor architectures. They show latency and throughput improvements across processor generations. In more recent architectures, these instructions also appear in "vectorized" (SIMD) versions that support processing up to 4 independent input streams in parallel. Additional instructions, namely `GF-NI`, have been added to x86-64 architectures and they can be useful as building blocks for symmetric key cryptography.

This paper briefly surveys the functional and performance characteristics of these crypto instructions and their usage for various constructions. It also describes some possible extensions to modes of operations with some desired properties, that the ecosystem can afford with the improved-throughput hardware support available in the modern processor architectures.

1. Preliminaries

The 128-bit block cipher AES was standardized by the National Institute of Standards and Technology (NIST) of the USA in 2001, and is specified in [FIPS197]. Its worldwide adoption made it the default block cipher of choice, and as such, its security and performance are very significant to the digital world. NIST has recently published a 20-years review of AES [Mou21]. This report points out the huge impact of AES, the strong support it has in the cryptographic community, and summarizes the results of 20 years of cryptanalysis research. Basically, AES remains unscathed for its intended usages, especially from all practical viewpoints. NIST's report also mentions (p. 1, [Mou21]) that:

> "Currently, virtually all modern 64-bit processors have native instructions for AES, which includes any recent 64-bit desktop or mobile device."

These native instructions are known as "AES-NI" and they were introduced with an additional processor instruction that is named "`PCLMULQDQ`".

AES is a permutation of $\{0, 1\}^{128}$, indexed by a cipher key (K) of length |K| =128, 192, or 256 bits. It encrypts a 128-bit plaintext block (X) under K by expanding K into a sequence of nr (nr =10, 12, 14 for |K| =128, 192, 256, respectively) additional 128-bit round keys (RK). The key schedule is RK [j], j=0, 1, …, nr, where RK [0] holds the first 128 bits of K. A 128-bit block is processed by a sequence of 44, 52, 60 (respectively to |K|) successive transformations described by the following pseudocode.

```
01    state = X
02    AddRoundKey(state, RK[0])
03    for j = 1 step 1 to nr-1 // nr=10,12,14
04      SubBytes(state)                          09    SubBytes(state)
05      ShiftRows(state)                         10    ShiftRows(state)
06      MixColumns(state)                        11    AddRoundKey(state, RK [nr])
07      AddRoundKey(state, RK[j])                12 // ciphertext = state
08    end for
```

The sub-sequence of 4 transformations in lines 04-08 is called an "AES round" (encryption) and the sub-sequence of 3 transformations in lines 09-11 is called an "AES last round" (encryption). Decryption is analogous, using the inverse transformations, an "AES round" (decryption) and an "AES last round" (decryption).

In the following, I briefly explain the AES-NI and PCLMULQDQ instructions, their features, and usages.

2. The AES-NI and PCLMULQDQ processor instructions

AES-NI is a set of six processor instructions (see definitions in [Gue09, Gue10]). Two instructions (AESIMC and AESKEYGENASSIST) support the AES key expansion and four instructions (AESENC, AESENCLAST, AESDEC, and AESDELAST) support the AES encryption and decryption flows. They were designed to provide full hardware support through the Instruction Set Architecture (ISA) for software that executes AES encryption or decryption, in all modes of operation. The functional description of AESENC and AESENCLAST is the following. Two 128-bit values (operands), X, Y, are input, and the 128-bit output is:

$$\text{AESENC}\ (X, Y) = \text{MixColumns}\ (\text{SubBytes}\ (\text{ShiftRows}\ (X))) \oplus Y$$

$$\text{AESENCLAST}\ (X, Y) = \text{SubBytes}\ (\text{ShiftRows}\ (X)) \oplus Y$$

Encryption of X under the key K, expanded to the sequence of round keys RK[i], i=0, …, nr, is the output of the iteration:

```
S = X ⊕ RK[0];
for j from 1 to (nr-1)
    S = AESENC (S, RK[j]);
end do;
S = AESENCLAST (S, RK[nr]);
```

Analogous definitions and flows are associated with AESDEC, AESDELAST and AES decryption. The following (assembly) code snippet illustrates AES encryption using AES-NI.

```
; Snippet 1.
; AES128 encryption (unrolled). Intel Syntax assembly: destination is the first operand.
; Registers xmm0-xmm10 hold round keys 0-10. Input (output) data is in xmm15.
    pxor xmm15, xmm0      ; Round 0 (Whitening step)
    aesenc xmm15, xmm1    ; Round 1          aesenc xmm15, xmm6      ; Round 6
    aesenc xmm15, xmm2    ; Round 2          aesenc xmm15, xmm7      ; Round 7
    aesenc xmm15, xmm3    ; Round 3          aesenc xmm15, xmm8      ; Round 8
    aesenc xmm15, xmm4    ; Round 4          aesenc xmm15, xmm9      ; Round 9
    aesenc xmm15, xmm5    ; Round 5          aesenclast xmm15, xmm10 ; Round 10
```

PCLMULQDQ is an instruction (see definition in [GK08a]) that computes the product (carry-less multiplication; denoted here by "●") of two binary polynomials of degree 63 (encoded as two 64-bit operands), which is a polynomial of degree 126 (encoded as a 128-bit operand with a zero most significant bit). The instruction takes two 128-bit operands, X1, X2, and a constant ("immediate") imm8 with 4

legitimate values (00, 01, 10 and 11, encoded as a byte). X1 is viewed as the concatenation of two binary polynomials of degree 63, P10 and P11, and X2 as the concatenation of two binary polynomials, P20 and P21. The instruction returns one of the polynomial products P10 ● P20, P11 ● P20, P10 ● P21 or P11 ● P21, where the selection is made by the value of imm8, and places it into the destination register.

AES-NI and `PCLMULQDQ` features. A most important property of AES-NI / `PCLMULQDQ`, which is key to their global adoption, is that their flexible architectural definition ([Gue09, Gue10, GK08, GK10]) makes them a tool with multiple usages:

- Simple software flows that use them can execute AES encryption and decryption with all standard key lengths (128/192/256 bits), in all standard modes of operation such as CTR, CBC, and all standard AEAD schemes such as AES-GCM, AES-CCM and OCB.
- AES-NI can be used for more than just the standard AES. For example, AES-NI can be used to implement the 256-bit block cipher Rijndael-256 (AES is a subset of the Rijndael proposal and is defined only for 128-bit blocks).
- Combinations of AES-NI and `PSHUFB` (byte shuffle instruction) can isolate the AES transformations MixColumns, SubBytes, ShiftRows and their inverses. This can be used, together with an equivalent reformulation of the AES key expansion (in terms of shuffles `AESENC` and `AESENCLAST`), for efficient execution of the AES key expansion (see [GLNP18]). This is useful for scenarios where short messages are encrypted under different keys and the relative cost of key expansion becomes more pronounced.

Throughput and latency. The difference between these two notions, and its effect on the usage of AES-NI is an important observation. If the latency of `AESENC`/`AESENCLAST` is L cycles, and XOR has a latency of 1 cycle, then AES encryption (with a 128-bit key) of one block would consume 1+10*L cycles (see Snippet 1). Since the block size is 16 bytes, the resulting processing rate is (0.0625 + 0.625*L) cycles per byte (cpb). For example, if L=4 (as in the latest processor generations), this amounts to 2.56 cpb. However, modern x86-64 processors have out-of-order features and the AES-NI hardware is fully pipelined. Therefore, the above performance can be improved when multiple independent blocks are encrypted, and software pipelining methodology is deployed. If L independent blocks are processed, *and* the code *interleaves* the AES rounds across the L states, a processor (with one microarchitectural "unit" for AES) can dispatch an `AESENC` (or `AESENCLAST`) instruction every cycle. This leads to a maximal *theoretical* processing rate of 10/16 = 0.625 cpb. Approaching this maximal rate in a real encryption mode of operation requires tuned optimized code because the processing involves additional operations on top of just the AES rounds. Consequently, it is possible to process *parallelizable* modes of operation such as counter mode (CTR) and CBC decryption at a much higher rate than serial modes such as CBC encryption. Note that prior to the introduction of AES-NI, optimized AES software used lookup tables and therefore the difference in software performance of AES serial and parallelizable modes was insignificant. Indeed, around 2010 (and a few years after) the dominant mode of encryption was the serial CBC mode (e.g., in TLS 1.1 and OS-based disk encryption). Interestingly, it is possible to speed up even the serial CBC encryption of multiple independent streams. This is done by interleaving AES rounds as shown in Snippet 2 for processing 4 streams. The number of streams required to fill the pipeline is L, so this code is optimized for processors where `AESENC`/`AESENCLAST` is 4 cycles. The performance of this implementation approaches the theoretical limit.

Snippet 2. CBC encryption of 4 messages, under 4 different keys.
The key schedules are pre-computed in the array KS.

```
 ***************************************
  .macro ROUND_interleaved i
    vaesenc \i*16+176*0(KS), STATE0, STATE0
    vaesenc \i*16+176*1(KS), STATE1, STATE1
    vaesenc \i*16+176*2(KS), STATE2, STATE2
    vaesenc \i*16+176*3(KS), STATE3, STATE3
  .endm
  ;
  ***************************************
  ; Round - (Whitening) and CBC XOR
  vxorps  16*0(INP), STATE0, STATE0
  vxorps  0*16+176*0(KS), STATE0, STATE0
  vxorps  16*1(INP), STATE1, STATE1
  vxorps  0*16+176*1(KS), STATE1, STATE1
  vxorps  16*2(INP), STATE2, STATE2
  vxorps  0*16+176*2(KS), STATE2, STATE2
  vxorps  16*3(INP), STATE3, STATE3
  vxorps  0*16+176*3(KS), STATE3, STATE3
  vxorps  16*4(INP), STATE4, STATE4
```

```
  ; First 9 AES rounds (interleaved)
  lea 16*8(INP), INP
  ROUND_interleaved 1
  ROUND_interleaved 2
  ROUND_interleaved 3
  ROUND_interleaved 4
  ROUND_interleaved 5
  ROUND_interleaved 6
  ROUND_interleaved 7
  ROUND_interleaved 8
  ROUND_interleaved 9
   Last round (interleaved)
  vaesenclast 10*16+176*0(KS), STATE0, STATE0
  vaesenclast 10*16+176*1(KS), STATE1, STATE1
  vaesenclast 10*16+176*2(KS), STATE2, STATE2
  vaesenclast 10*16+176*3(KS), STATE3, STATE3
 ; results in STATE0, STATE1, STATE2, STATE3
```

`PCLMULQDQ` as a building block. Originally, the main motivation for introducing the instruction `PCLMULQDQ` was to speed up multiplication in GF (2^{128}) with the reduction polynomial $P = x^{128} + x^7 + x^2 + x + 1$ that is used for AES-GCM. Of course, as a generic arithmetical instruction, `PCLMULQDQ` itself is agnostic to P and can be used in multiple ways. Snippet 3 below demonstrates GF (2^{128}) multiplication (mod P) using `PCLMULQDQ`. The code uses the schoolbook (carry-less) multiplication and Gueron's fast reduction algorithm [Gue13]. The code (assembly; AT&T syntax) is taken from the AES-GCM-SIV implementation in https://github.com/Shay-Gueron/AES-GCM-SIV (which is also integrated into BoringSSL).

```
 ; Snippet 3.
  .align  16
 poly:
  .quad 0x1, 0xc200000000000000
  #Used by _GFMUL     #
  .set RES, %xmm0     #
  .set H, %xmm1       #
  .set TMP1, %xmm2    #
  .set TMP2, %xmm3    #
  .set TMP3, %xmm4    #
  .set TMP4, %xmm5    #
 # RES = _GFMUL(RES, H)
 # a = RES; b = H; res = RES
 # intermediate TMP1,TMP2,TMP3,TMP4
 # __m128i _GFMUL(__m128i A, __m128i B);
  .type _GFMUL,@function
  .globl _GFMUL
 _GFMUL:
      vpclmulqdq  $0x00, H, RES, TMP1
      vpclmulqdq  $0x11, H, RES, TMP4
      vpclmulqdq  $0x10, H, RES, TMP2
      vpclmulqdq  $0x01, H, RES, TMP3
      vpxor       TMP3, TMP2, TMP2
      vpslldq     $8, TMP2, TMP3
      vpsrldq     $8, TMP2, TMP2
      vpxor       TMP3, TMP1, TMP1
      vpxor       TMP2, TMP4, TMP4
```

```
      vpxor       TMP3, TMP2, TMP1
      vpshufd     $78, TMP1, TMP3

      vpclmulqdq  $0x10, poly(%rip), TMP1, TMP2
      vpshufd     $78, TMP1, TMP3
      vpxor       TMP3, TMP2, TMP1
      vpclmulqdq  $0x10, poly(%rip), TMP1, TMP2
      vpshufd     $78, TMP1, TMP3
      vpxor       TMP3, TMP2, TMP1

      vpxor       TMP4, TMP1, RES
      ret
 .size _GFMUL, .-_GFMUL
```

AES-NI and `PCLMULQDQ` performance across processor generations. By now, AES-NI and `PCLMULQDQ` are ubiquitous and are part of the ISA of all Intel processors, AMD processors and ARM processors (from V7). The impact is twofold: a) their contribution to software performance of encryption and authentication significantly changed the way that the ecosystem perceives encryption overheads; b) they removed the dependency of AES software on lookup-tables, which spawned various cache-access side channel attacks and concern with regards to the security of AES software implementations. Optimized hardware implementation of the underlying circuits (e.g., [MSAK10]) improved the microarchitecture underlying AES-NI / `PCLMULQDQ` instructions. This has reduced their latency from 8 to 4 cycles across processor generations with 1 cycle throughput. To facilitate proliferation of usage (advocated in [KKG+10]), optimized code implementations were contributed to open-source libraries (OpenSSL, BoringSSL, NSS). In more recent architectures, AES-NI and `PCLMULQDQ` are dispatched from separate execution ports and this allows for better pipelining of CTR encryption and GHASH computations for AES-GCM (and AES-GCM-SIV decryption). More speed ups to AES-GCM were gained by a mathematical reformulation of GHASH. This, together with a judicious use of the ISA, facilitates aggressive optimization of AES-GCM [Gue13].

Impact. In ~10 years, these multiple contributions improved the AES-GCM throughput (with a 128-bit key) from ~22 cpb prior to the instructions, and hence *slower* than the dominating AES-CBC with HMAC-SHA1 alternative at the time, to ~0.65 cpb (on Intel's microarchitecture code name "Skylake"). With this, AES-GCM authenticated encryption is practically as fast as only the CTR encryption. The effect was that the ubiquitous TLS communications usage of AES-CBC-HMAC-SHA1 transitioned to the faster and more secure AES-GCM mode on all server and client platforms with AES/`PCLMUL` support (small devices without this hardware support use ChaCha-Poly1305). Most of today's networking traffic inside and outside critical datacenters is processed with AES-GCM. For encryption-only modes, OS-based disk encryption (e.g., Windows Bitlocker) replaced the originally used CBC mode with the parallelizable XTS mode. Today, virtually all the open-source (and proprietary) crypto libraries support implementations based on AES-NI and `PCLMULQDQ` and the performance advantage of parallelization is well understood and leveraged. Symmetric key encryption and authentication benchmarks rely on, or at least quote, the numbers with AES-NI and `PCLMULQDQ` when relevant (e.g., with AES-GCM and AES-GCM-SIV). The performance of the standard AEAD OCB3 [KR21] approaches that of CTR encryption only with the use of AES-NI and efficient software pipelining (`PCLMULQDQ` is not used for this mode). To illustrate, Table 1 shows the effect of algorithms, software optimization and microarchitectural changes on the (optimized) software performance with AES-NI and `PCLMULQDQ`, across (Intel) processor generations (2010-2018).

Table 1. Software performance of AES standard modes across (Intel) processor generations, measured in cycles per byte over 8KB buffers.

Mode / processor	CTR	XTS	CBC decryption	CBC Encryption (serial)	CBC Encryption (multiple streams)	AES-GCM
Sandy Bridge	0.76	1.21	0.8	5.05	0.9	2.75
Haswell	0.64	0.7	0.65	4.41	0.8	1.02
Broadwell	0.64	0.7	0.65	4.41	0.8	0.76
Skylake	0.63	0.63	0.62	2.65	0.64	0.65
Cannon Lake	0.41	0.5	0.36	2.56	0.37	0.5

Innovative and creative use of AES-NI / `PCLMULQDQ`. Since their inception, the performance benefits (especially in pipelined sequences) and flexible usability of AES-NI and `PCLMULQDQ` motivated multiple innovative designs. The following brief description illustrates some of the designs that use the AES round as

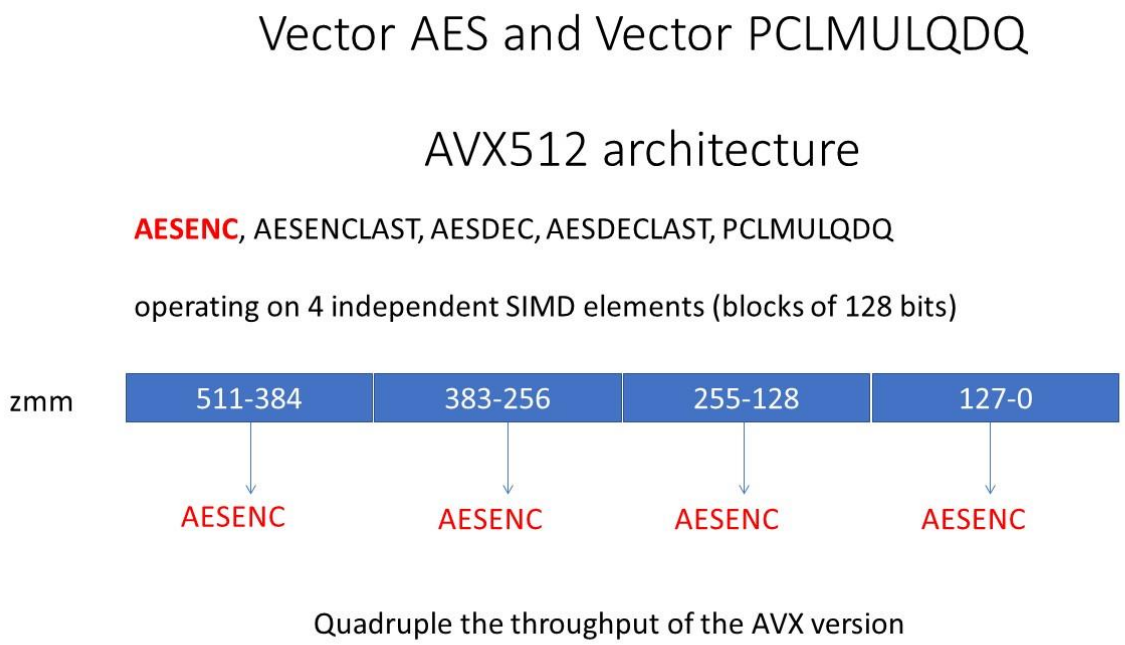
a building block. One example is the reduced-round (4 and 6 rounds) AES as a component in AEZ [HKR15] and LmD [BDMN16]. AES components appeared in some of the SHA-3 proposals [BBGR09], and analysis of efficient hashing constructions using appears in [BOS11]. AES-NI instructions have apparent impact on the CAESAR competition (https://competitions.cr.yp.to/caesar.html) where almost all the authenticated cipher winning proposals (e.g., AEGIS-128, OCB, Deoxys-II) pipeline AES or AES elements for performance. `AESENC` inspired the use of a single AES round as a building block, and examples include the short-input keyed hash Haraka [KLMR16] (used as a component in the SPHINCS+ Post-Quantum signature scheme proposal). The cryptographic permutation Simpira [GM16b] of 128×b bits (b is a parameter) uses two unkeyed AES rounds as a building block (see [GM17] for its use in SPHINCS+).
`PCLMULQDQ` has been used for binary ECC [GK08b], for CRCs and fast error detection, and for high degree polynomial multiplication in some post quantum KEM proposals (https://github.com/awslabs/bike-kem). The universal hash function POLYVAL that is used with the nonce misuse resistance scheme AES-GCM-SIV [GLL19] is defined with a special optimization for Little-Endian architectures (that can be leveraged with the `PCLMULQDQ` instruction and make the computations faster than computing GHASH on the same input).

2.1 Abundant AES: vectorization of AES-NI and `PCLMULQDQ`

Following the successful proliferation of AES-NI and `PCLMULQDQ` usage, I promoted the introduction of *vectorized* versions for these instructions, which I call here "vector AES" and "vector `PCLMULQDQ`". These are explained in [DGK19] and [DGK18a]. Such instructions are available in processors from 2019 (starting from Intel's "Ice Lake"). These instructions extend AES-NI and `PCLMULQDQ` to the AVX2 and AVX512 architectures, allowing the related functionality to be executed (in parallel) on 2 or on 4 "elements", stored in a wide register (of 256 or 512 bits). For brevity, I focus here only on the AVX512 version as illustrated below for vectorized AESENC with the call syntax:

```
vaesenc zmm1, zmm1, zmm2
```

When it is intended for AES computations, the 512-bit register zmm1 holds 4 AES states, and zmm2 holds 4 round keys (possibly from different main keys).



At a first glance, it seems that the vectorized instructions would automatically quadruple the observed performance of cryptographic workloads, but reality is more subtle. The achievable performance depends on the scheme (mode of operation) and on the characteristics of the usage (e.g., expected lengths of the input

data and parallelizability of the processing). Extracting the full potential of the new architectural offering requires careful software optimization. To illustrate the performance subtleties, consider the following experiment where the same AES-CTR mode code is run on two microarchitectures, using legacy (not vectorized) AES-NI.

The code uses the one-stream AES-NI (i.e., on an AVX architecture) and pipelines 8 blocks in parallel. The following (assembly) macro illustrates the 8-wise pipelining of an AES round (over 8 counters)

```
-------------------------------------------------------------------------------------
.macro AES_ROUND i
 vmovdqu  \i*16(%r9), TMP5
 vaesenc  TMP5, CTR0, CTR0
 vaesenc  TMP5, CTR1, CTR1
 vaesenc  TMP5, CTR2, CTR2
 vaesenc  TMP5, CTR3, CTR3
 vaesenc  TMP5, CTR4, CTR4
 vaesenc  TMP5, CTR5, CTR5
 vaesenc  TMP5, CTR6, CTR6
 vaesenc  TMP5, CTR7, CTR7
.endm
-------------------------------------------------------------------------------------
```

The measured performance of this CTR encryption when processing 4KB on the Skylake processor is ~0.66 cpb. Note that the 8-wise software pipelining fills the processor's pipe because on this processor, the latency of `AESENC` (and `AESENCLAST`) is 4 cycles and the throughput is 1. The theoretical throughput of AES (128-bit key) is 10/16=0.625 cpb. So, the observed performance is within ~1.5% of the theoretical maximum. This is due to some additional operations on top of the pure AES rounds in CTR mode. In this implementation, the counter block is stored in a 128-bit register, the IV populates 96 bits and the 32-bit counter is incremented in a Big-Endian style on its Little-Endian storage (in the register). Finally, some load/store overheads take some toll. The same code is run on a different processor (namely Cannon Lake) that has `AESENC` and `AESENCLAST` on 2 execution ports (supported by 2 AES hardware units). The observed performance is 0.43 cpb, where in theory, doubling `AESENC` throughput could double the performance to potentially ~0.32 cpb.

3. GF-NI: more ISA "vocabulary" for cryptographic constructions

GF-NI (Galois Field New Instructions) are new instructions that I defined for Intel's x86-64 ISA. They first appeared in real silicon in 2019 in the "Ice Lake" processor (see demonstrations in [DGK18b] for fast computations of the Reed-Solomon codes). These instructions have AVX, AVX2, and AVX512 versions for registers of 128, 256 and 512 bits (as well as an SSE version). The GF-NI set includes the instructions `VGF2P8MULB`, `VGF2P8AFFINEQB`, and `VGF2P8AFFINEINVQB` (hereafter `MULB`, `AFFINEB`, and `AFFINEINVB`, respectively, for short). These are byte-wise operations that manipulate data elements stored in (up to 64) bytes of one or two registers. The bytes are viewed as elements of GF (2^8). The instruction `MULB` multiplies GF (2^8) elements held in the bytes of two registers (operands) and places the results in the bytes of a destination register. For this instruction, GF (2^8) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$. This definition binds the architecture (and the underlying microarchitecture) to a specific polynomial and seems to be restrictive. However, the instructions `AFFINEB` and `AFFINEINVB` compute the following affine transformations, as bytewise operations on the byte elements of their operands, respectively: $A \cdot x + b$ and $A \cdot inv(x) + b$. Here, A is an 8×8-bit matrix, x and b are 8-bit vectors, "$\cdot$" denotes matrix-vector multiplication over GF (2), and inv(x) denotes inversion in GF (2^8) with the above representation. The matrix A is encoded in one of the operands, and the vector b is encoded in an immediate byte.

For example, consider the matrix A and vector b that are part of the AES Sbox definition:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

To encode this transformation as operands of the AFFINEINVB instruction, the rows of the matrix are encoded as a hexadecimal value (for example, the first row is 11111000b = 0xf8, and the second row is 01111100b=0x7c). Finally, the vector is encoded in the immediate byte: 01100011b = 0x63.
This way, the instruction

```
VGF2P8AFFINEINVQB xmm1, xmm2, xmm3, 0x63
```

Computes A · inv(x) + b (affine transformation over an inverse). For example, the operands

```
xmm2 = 0x31,0x69,0x8f,0xd9,0xd6,0xda,0x56,0xa9,0x8d,0x67,0x43,0x5d,0x42,0x71,0x57,0xce

xmm3 = 0xf8,0x7c,0x3e,0x1f,0x8f,0xc7,0xe3,0xf1,0xf8,0x7c,0x3e,0x1f,0x8f,0xc7,0xe3,0xf1
```

produce the output (placed in the destination register xmm1):

```
xmm1 = 0xc7,0xf9,0x73,0x35,0xf6,0x57,0xb1,0xd3,0x5d,0x85,0x1a,0x4c,0x2c,0xa3,0x5b,0x8b
```

which is the computation of the AES Sbox.
An example for the affine transformation A · x + b is:

```
   VGF2P8AFFINEQB xmm1, xmm2, xmm3, 0x21
   xmm2 = 0x41,0x75,0xb7,0xa3,0x53,0xaa,0x22,0x7a,0x73,0x98,0x17,0x63,0x8d,0x39,0x06,0x0f
   xmm3 = 0x44,0x73,0x9d,0xef,0x04,0x7c,0x37,0x7d,0x44,0x73,0x9d,0xef,0x04,0x7c,0x37,0x7d
output:
   xmm1 = 0x87,0x08,0xbf,0x67,0xb2,0x21,0x24,0xa3,0xe5,0x43,0xd8,0x82,0xe8,0x57,0xcc,0x8a
```

Unlike AFFINEINVB and MULB, the instruction AFFINEB is *agnostic* to the representation of the GF (2^8). GF-NI instructions have the following properties:

- AFFINEINVB can compute the inverse in GF (2^8) with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$ if A is the identity matrix and b=0.
- Since all representations of GF (2^8) are isomorphic, AFFINEB can be used to convert GF (2^8) elements to any other representation (and back) by placing the proper transformation matrix as A.
- Since the composition of affine transformations is also an affine transformation, AFFINEINVB can compute Sbox values of the form M · inv(x) + n or the form inv (M · y + n) in any GF (2^8) representation.
- It is possible to compute an Sbox of the form A · inv (M · x + n) + b by invoking AFFINEB followed by AFFINEINVB (for any representation of GF (2^8)).

This leads to the following observations:

- It is possible to execute AES with no lookup tables without AES-dedicated instructions (although with some performance degradation compared to using AES-NI directly). This can be leveraged in geographies where the AES-NI instructions are disabled.
- It is possible to use GF-NI for efficiently computing the different Sbox constructions of the Chinese block cipher SM4, Japanese block cipher Camellia, and Chinese stream cipher ZUC (they operate with different representations of GF (2^8) and different matrices/vectors).
- GF-NI instructions offer a flexible vocabulary of (bytewise) algebraic functionality with highly nonlinear bitwise complexity, which are fast to compute (without lookup tables). These can be used for implementing Sbox designs with potentially preferred properties (see e.g., [CHZ+11], [NST20]).

AFFINEB can be used for new constructions. Note that there are 2^{64} binary 8 × 8 matrices and from these,

$$(2^8\text{-}1)*(2^8\text{-}2)*(2^8\text{-}4)*(2^8\text{-}8)*(2^8\text{-}16)*(2^8\text{-}32)*(2^8\text{-}64)*(2^8\text{-}128) = 5348063769211699200 \sim 2^{62.2}$$

are invertible (in GF (2)) matrices. Thus, with probability ~29% `AFFINEB` with a uniform random binary 8 × 8 matrix (and any offset 8-bit vector) will specify an *invertible* transformation. In particular, all the 8! binary 8 × 8 permutation matrices (i.e., a single set bit in each row and each column, and 0 elsewhere) can be used with `AFFINEB` in order to compute any possible bit permutation inside a byte. Examples include bit reversal, nibble swapping, and rotation. This can be used for adding invertible transformations (e.g., pseudorandomly) as input to `AFFINEB`, as a supplemental diversification step in a design. `AFFINEB` can also be used for non-invertible bitwise operations (e.g., bit-test, bit duplication, bit selection). The conclusion is that GF-NI can be used for studying a full range of new cryptographic primitives. Creative use, potentially with other components of modern ISAs (e.g., ternary logic), and parallelization, can lead to defining new interesting designs.

4. Leveraging crypto instructions in modern architectures for new constructions with improved properties

4.1 Leveraging the "abundant AES"

With vectorized AES-NI, the cryptographic ecosystem can afford to define constructions with improved desired properties. The following example is an illustration.

Consider the standard CTR confidentiality mode with a random 96-bit IV per message. A message is encrypted by XOR-ing its blocks with the pseudorandom stream generated by computing AES (K, IV || Ctr) where Ctr is a running counter and K is the encryption key. The security of CTR depends critically on uniqueness of the IVs. If Q messages are encrypted, the probability of encountering an IV collision is $\sim Q^2/2^{129}$, and this limits the number of allowed encryptions in K. For example, if 2^{-32} is the tolerable collision probability, K can be used for only Q ~2^{32} encryptions before it must be rotated. Such key rotation rate may seem expensive (and undesired) for encryption at scale (e.g., at cloud scale). However, note that with an abundant AES, offered by vectorized AES-NI, applications can "afford" to invoke more than one AES computation per block. This allows for choosing longer random IVs to reduce the collision probability and thus extend the lifetime of the key. A loose example is a "double CTR" mode is shown below.

Notation: For integer j s.t. $0 \le j < 2^p$ denote the p-bit encoding of the binary representation of j by ${}^{p}[j]_2$ (e.g., ${}^{8}[13]_2$ = 00001101).
For a string S with bitlength denoted |S|, and an integer $t \le |S|$, denote the t most significant bits of S by msb_t (S).

Algorithm DoubleCTR (K, $N^{\$}$, M)
Key K (|K| = keylen)
Input: message M ∈ $\{0, 1\}^*$ s.t., $|M| < 2^{45}$ (blocks)
w = ceil(|M|/128)
N ←$ $\{0, 1\}^{160}$ // use a 160 bits uniform random nonce
N1 = N [159:80] // split N into 2 halves N1, N2
N2 = N [79:0]
for j from 1 to w
    U1 [j] = AES (K, N1 || 0 || 00 || ${}^{45}[j]_2$)
    U2 [j] = AES (K, N1 || 0 || 01 || ${}^{45}[j]_2$)
    U3 [j] = AES (K, N2 || 1 || 10 || ${}^{45}[j]_2$)
    U4 [j] = AES (K, N2 || 1 || 11 || ${}^{45}[j]_2$)
    Pad [j] = U1 [j] ⊕ U2 [j] ⊕ U3 [j] ⊕ U4 [j]
end
output C = M ⊕ $msb_{|M|}$(Pad [1] || Pad [2] || … || Pad [w])

4.2 Adding a preamble to a nonce-based scheme

A nonce-based "Preamble" is a generalization of the Derive-Key schemes described by Gueron and Lindell [GL17]. It starts from a given nonce-based scheme. The Preamble is some PRF that uses a main key (K), takes a nonce (N) and derives some pseudorandom values from K and N (or possibly K alone). Subsequently, the underlying scheme is invoked with the derived values and the regular input that would have been used directly.

The Derive-Key paradigm of [GL17] derives a fresh key for every invocation of a nonce-based encryption or AEAD scheme to process a message M under key K and nonce N. This template can be described as a two-step sequence:

K' ← Preamble (K, N)
(C, Tag) = Scheme (K', N, M)
Output: C, Tag, key-commitment-string

The paper [GL17] demonstrates the effect of adding this Preamble in several cases. One example is the CTR mode with a non-repeating nonce as a counter of 96 bits. Here, adding the preamble reduces the distinguishing advantage upper bound of a CPA adversary and this allows for encrypting more data than with the plain CTR. However, in the random 96-bit nonce setting, the Preamble cannot increase the allowed number of messages, 2^{32}, for keeping the collision probability below 2^{-32}, because the composition does not change the nonce collision probability. The nonce misuse resistant AEAD AES-GCM-SIV is also analyzed [GL17]. This scheme uses a preamble to derive per-nonce encryption and hash keys, subsequently followed by invoking GCM-SIV+. AES-GCM-SIV is now RFC8542 [GLL19] and is already used in various scenarios.

In general, such constructions the Preamble should be a PRF with (very) good indistinguishability upper bounds for the number of calls that the usage is designed to support. The specific preamble used for AES-GCM-SIV is based on truncating a permutation (AES in this case) into half. This way, generating 128d bits of keying material requires 2d AES invocations. These invocations are over independent blocks so efficient pipelining keeps the derivation step quick. There are other Beyond-Birthday-Bound PRFs (e.g., CENC [Iwa06]) that require fewer invocations. However, the performance difference in software implementations is marginal.

5. Discussion

The paper provided a short survey on the performance that symmetric key encryption and authentication – standard and innovative - can achieve by using crypto instructions found today in most processors. It showed that these instructions have been utilized for multiple purposes and more creative options can be explored.

The following are a few options and enhancements that can be considered for the current portfolio of standards.

Standardize AES-GCM-SIV to leverage its nonce misuse resistance security feature. This AEAD is already an RFC (RFC8542 [GLL19]) and is already used in various scenarios in the industry. AES-GCM-SIV has implementations in multiple programming languages and is part of the cryptographic libraries BoringSSL and OpenSSL.

Standardize the nonce-based Preamble approach as an acceptable way to enhance standard schemes. Preambles can be defined in multiple ways that rely only on one universally accepted assumption:

> AES (with a uniform random secret key) is indistinguishable from a random permutation of $\{0, 1\}^{128}$, even for adversaries that are given a very large number of queries.

With this, there are multiple ways to leverage proven PRF constructions that are based on random permutations. These constructions can be executed efficiently with AES-NI, especially if they sample the permutations over independent blocks. It is possible to use the 128-bit block size AES and operate on a

nonce whose length is only a few bits less than 128. As an example, consider the Preamble

$$(K, N) \rightarrow \big(\text{AES}(K, N \,\|\, 00) \oplus \text{AES}(K, N \,\|\, 01)\big) \,\|\, \big(\text{AES}(K, N \,\|\, 10) \oplus \text{AES}(K, N \,\|\, 11)\big)$$

that takes a 126-bit nonce and derives a 256-bit subkey (to be subsequently used with an encryption algorithm, such as AES-GCM). In the randomized nonce scenario, a long nonce would reduce the nonce collision probability compared to a choice of 96-bit nonces. To get an even smaller nonce collision probability it is possible to define a Preamble that takes nonces with more than 128 bits (see [Gue22]). Note that a Preamble can be used for deriving more than just a fresh subkey. For example, for deriving (from either K or from K and N) a public string that is used as a key commitment (key identifier) for schemes that do not have this property (e.g., AES-GCM; see [GR17]).

(K', key-commitment-string) ← Preamble (K, N);
(C, Tag) = Scheme (K', N, M)
Output: C, Tag, key-commitment-string

5.1 Standardizing 256-bit block ciphers

The output of a 128-bit block cipher (a permutation of $\{0, 1\}^{128}$) is distinguishable from random after 2^{64} invocations. This limits the number of usages allowed with a given key. These limitations can be deferred if a wider block size is used. Therefore, a standardized 256-bit block cipher (permutation of $\{0, 1\}^{256}$) can be a useful primitive for new constructions. This leads to the search for a secure 256-bit block cipher (say, with a 256-bit key) that would run efficiently on modern platforms, and to the obvious question: how fast can such a cipher run (compared to AES256)?

The following paragraphs explore two options and provide some answers.

Rijndael-256. The original "Rijndael" proposal by Rijmen and Daemen included a definition for a 256-bit block size and a 256-bit key, but this variant was not standardized. Interestingly (see [Gue09, Gue10]), AES-NI can be used for executing Rijndael-256 (with no lookup tables). To assess the potential performance, note that Rijndael-256 calls `AESENC` 28 times (twice per round, over 14 rounds). Therefore, on a processor with a single AES unit, its maximal performance is 28/32 = 0.875 cpb. The vector AES architectures can potentially quadruple this throughput, but there are some overheads that need to be accounted for. This is explained in detail in [DG22] and the authors report performance of 0.27 cpb on x86-64 platforms with vector AES-NI instructions. This shows that from a performance viewpoint, Rijndael-256 can is an efficient alternative.

Simpira. Simpira is a family of unkeyed permutations of 128b bits with an indifferentiability claim for up to 2^{128} queries [GM16b]. Its design is built for high throughput and motivated by the presence of `AESENC` (the only needed instruction). For b=2, Simpira can be a competitive cryptographic permutation primitive. It uses two unkeyed AES rounds as the underlying primitive, in a 15 round Feistel flow, as described in the following snippet (in C syntax):

```
#define C(i) _mm_setr_epi32(0x00^(i)^2, 0x10^(i)^2, 0x20^(i)^2, 0x30^(i)^2)
#define R(y,z,i){ \ z = _mm_aesenc_si128(_mm_aesenc_si128(y,C(i)), z); \}
void Simpira_b_2 (uint8_t* in, uint8_t* out) {
 __m128i x[2]; __m128i c0;
    x[0] = *(__m128i*)in;
    x[1] = *(__m128i*)(in+16);
    R(x[0],x[1], 1);    R(x[1],x[0], 2);
    R(x[0],x[1], 3);    R(x[1],x[0], 4);
    R(x[0],x[1], 5);    R(x[1],x[0], 6);
    R(x[0],x[1], 7);    R(x[1],x[0], 8);
    R(x[0],x[1], 9);    R(x[1],x[0],10);
    R(x[0],x[1],11);    R(x[1],x[0],12);
    R(x[0],x[1],13);    R(x[1],x[0],14);
    R(x[0],x[1],15);
    *(__m128i*)out = x[0];
    *(__m128i*)(out+16) = x[1];
}
```

Simpira can be used as a tweakable 256-bit block cipher C = Simpira (P ⊕ K × T) ⊕ K × T where P is the plaintext, K is the key, T is a nonzero tweak, and × represents GF (2^{256}) multiplication. Note that fixing T =1 degenerates the tweakable cipher into an Even-Mansour construction. The (non-tweakable) design involves 30 `AESENC` calls, so the maximal throughput is 30/32~0.94 cpb on a platform with a *single* AES unit. This throughput is closely achieved on the Skylake processor, if 4 Simpira tasks are parallelized, and code is optimized for pipelining. Platforms with vector AES-NI show the expected quadrupled speedups. From a latency viewpoint, such code outputs 4×256-bit outputs (128 bytes) in ~125 cycles. Note that this block cipher construction does not require key expansion, so it is suitable for usages that rotate their keys frequently, e.g., with a derived nonce-based key. It is worth mentioning that the Simpira 256-bit block cipher (with or without a tweak) can be used to instantiate a Preamble. This allows for easily supporting a long nonce (e.g., 224 bits).

Acknowledgements. This research was supported by NSF-BSF Grant 2018640, The Israel Science Foundation (grant No. 3380/19), The Center for Cyber Law and Policy at the University of Haifa, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

[FIPS197] National Institute of Standards and Technology (2001). Advanced Encryption Standard (AES). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 197. https://doi.org/10.6028/NIST.FIPS.197.

[BBGR09] Benadjila R., Billet O., Gueron S., Robshaw M.J.B. (2009) The Intel AES Instructions Set and the SHA-3 Candidates. In: *Matsui M. (eds) Advances in Cryptology – ASIACRYPT 2009. ASIACRYPT 2009*. Lecture Notes in Computer Science, vol 5912. Springer, Berlin, Heidelberg.

[BOS11] Bos J.W., Özen O., Stam M. (2011) Efficient Hashing Using the AES Instruction Set. In: *Preneel B., Takagi T. (eds) Cryptographic Hardware and Embedded Systems – CHES 2011. CHES 2011. Lecture Notes in Computer Science*, vol 6917. Springer, Berlin, Heidelberg.

[BDMN16] Bossuet, L., Datta, N., Mancillas-López, C., Nandi, M. (2016, November). ELmD: A Pipelineable Authenticated Encryption and Its Hardware Implementation. In *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3318-3331.

[CHZ+11] Cui, J., Huang, L., Zhong, H., Chang, C., Yang, W. (2011). An improved AES S-box and its performance analysis. *International Journal of Innovative Computing, Information and Control*, Vol 7 (pp. 2291–2302).

[DGK18a] Drucker, N., Gueron, S., & Krasnov, V. (2018, June). Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction. In *2018 IEEE 25th Symposium on Computer Arithmetic* (ARITH) (pp. 115-119). IEEE.

[DGK18b] Drucker, N., Gueron, S., & Krasnov, V. (2018, June). The comeback of Reed Solomon codes. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)* (pp. 125-129). IEEE.

[DGK19] Drucker, N., Gueron, S., & Krasnov, V. (2019). Making AES great again: the forthcoming vectorized AES instruction. In *16th International Conference on Information Technology-New Generations (ITNG 2019)* (pp. 37-41). Springer, Cham.

[DG22] Drucker, N., & Gueron, S. (2022). Software Optimization of Rijndael for Modern x86-64 Platforms. In *ITNG 2022 19th International Conference on Information Technology-New Generations* (pp. 147-153). Springer, Cham.

[GR17] Grubbs P., Lu J., Ristenpart T. (2017) Message Franking via Committing Authenticated Encryption. In: Katz J., Shacham H. (eds) Advances in Cryptology – CRYPTO 2017. CRYPTO 2017. Lecture Notes in Computer Science, vol 10403. Springer, Cham.

[Gue09] Gueron, S. (2009, February). Intel's new AES instructions for enhanced performance and security. In *International Workshop on Fast Software Encryption* (pp. 51-66). Springer, Berlin, Heidelberg.

[Gue10] Gueron, S. (2010). Intel advanced encryption standard (AES) instructions set. *Intel White Paper, Rev*, *3*, 1-94 https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[Gue13] Gueron, S. (January, 2016). AES-GCM for Efficient Authenticated Encryption – Ending the Reign of HMAC-SHA-1?. *In Workshop on Real-World Cryptography (Real World Crypto)* https://crypto.stanford.edu/RealWorldCrypto/ (2013).

[Gue22] Gueron, S. (2022). Counter Mode for Long Messages and a Long Nonce. *In: Dolev, S., Katz, J., Meisels, A. (eds) Cyber Security, Cryptology, and Machine Learning. CSCML 2022.* Lecture Notes in Computer Science, vol 13301. Springer, Cham.

[GK08a] Gueron, S., & Kounavis, M. (2008). Carry-less multiplication and its usage for computing the GCM mode. *White Paper, Intel Corporation*. https://www.intel.ph/content/dam/www/public/us/en/documents/white-papers/carry-less-multiplication-instruction-in-gcm-mode-paper.pdf

[GK08b] Gueron, S., & Kounavis, M. (2008, April). A technique for accelerating characteristic 2 elliptic curve cryptography. In *Fifth International Conference on Information Technology: New Generations (ITNG 2008)* (pp. 265-272). IEEE.

[GK10] Gueron, S., & Kounavis, M. (2010). Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Information Processing Letters*, *110*(14-15), 549-553.

[GL17] Gueron, S., & Lindell, Y. (2017, October). Better bounds for block cipher modes of operation via nonce-based key derivation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1019-1036).

[GLNP18] Gueron, S., Lindell, Y., Nof, A., & Pinkas, B. (2018). Fast garbling of circuits under standard assumptions. *Journal of Cryptology*, *31*(3), 798-844.

[GLL19] Gueron, S., Langley, A., & Lindell, Y. (2019). AES-GCM-SIV: Nonce misuse-resistant authenticated encryption. RFC 8452. https://datatracker.ietf.org/doc/html/rfc8452

[GM16a] Gueron, S., & Mathew, S. (2016, July). Hardware implementation of AES using area-optimal polynomials for composite-field representation GF $(2^4)^2$ of GF (2^8). In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)* (pp. 112-117). IEEE.

[GM16b] Gueron, S., & Mouha, N. (2016, December). Simpira v2: A family of efficient permutations using the AES round function. In *International Conference on the Theory and Application of Cryptology and Information Security* (pp. 95-125). Springer, Berlin, Heidelberg.

[GM17] Gueron, S., & Mouha, N. (2017). SPHINCS-Simpira: Fast Stateless Hash-based Signatures with Post-quantum Security. In *Cryptology ePrint Archive, Report* 2017/645 https://ia.cr/2017/645

[HKR15] Hoang V.T., Krovetz T., Rogaway P. (2015). Robust Authenticated-Encryption AEZ and the Problem That It Solves. In: *Oswald E., Fischlin M. (eds) Advances in Cryptology -- EUROCRYPT 2015. EUROCRYPT 2015*. Lecture Notes in Computer Science, vol 9056. Springer, Berlin, Heidelberg.

[KR21] Krovetz, T., Rogaway, P. The Design and Evolution of OCB. *J Cryptol* 34, 36 (2021). https://doi.org/10.1007/s00145-021-09399-8

[Iwa06] Iwata, T. (2006). New blockcipher modes of operation with beyond the birthday bound security. *In: Fast Software Encryption, 13th International Workshop, FSE 2006*. Lecture Notes in Computer Science, vol. 4047, pp. 310–327. Springer (2006)

[KLMR16] Kölbl, S., Lauridsen, M. M., Mendel, F., & Rechberger, C. (2016). Haraka v2–efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology*, 1–29.

[KKG+10] Kounavis, M. E., Kang, X., Grewal, K., Eszenyi, M., Gueron, S., & Durham, D. (2010). Encrypting the internet. *ACM SIGCOMM Computer Communication Review*, *40*(4), 135-146.

[MSAK10] Mathew, S., Sheikh, F., Agarwal, A., Kounavis, M., Hsu, S., Kaul, H., ... & Krishnamurthy, R. (2010, June). 53Gbps native GF $(2^4)^2$ composite-field AES-encrypt/decrypt accelerator for content-protection in 45nm high-performance microprocessors. In *2010 Symposium on VLSI Circuits* (pp. 169-170). IEEE.

[Mou21] Mouha, N. (2021, July). Review of the Advanced Encryption Standard. NISTIR 8319, https://csrc.nist.gov/publications/detail/nistir/8319/final

[NST20] Nitaj, A., Susilo, W., Tonien, J. (2020). A New Improved AES S-box With Enhanced Properties. In *Cryptology ePrint Archive*, Report 2020/1597 https://ia.cr/2020/1597

[Rog11] Rogaway, P. (2011). Evaluation of Some Blockcipher Modes of Operation. https://www.cs.ucdavis.edu/~rogaway/papers/modes.pdf