SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures

Armando Faz-Hernández
Institute of Computing
University of Campinas
Campinas, São Paulo, Brazil
armfazh@ic.unicamp.br

Julio López
Institute of Computing
University of Campinas
Campinas, São Paulo, Brazil
jlopez@ic.unicamp.br

Ana Karina D. S. de Oliveira
Federal University of Mato Grosso Do Sul (FACOM-UFMS)
Campo Grande MS, Brazil
anakarina@facom.ufms.br

ABSTRACT

The latest processors have included extensions to the instruction set architecture tailored to speed up the execution of cryptographic algorithms. Like the AES New Instructions (AES-NI) that target the AES encryption algorithm, the release of the SHA New Instructions (SHA-NI), designed to support the SHA-256 hash function, introduces a new scenario for optimizing cryptographic software. In this work, we present a performance evaluation of several cryptographic algorithms, hash-based signatures and data encryption, on platforms that support AES-NI and/or SHA-NI. In particular, we revisited several optimization techniques targeting multiple-message hashing, and as a result, we reduce by 21% the running time of this task by means of a pipelined SHA-NI implementation. In public-key cryptography, multiple-message hashing is one of the critical operations of the XMSS and XMSS^{MT} post-quantum hash-based digital signatures. Using SHA-NI extensions, signatures are computed 4× faster; however, our pipelined SHA-NI implementation increased this speedup factor to 4.3×. For symmetric cryptography, we revisited the implementation of AES modes of operation and reduced by 12% and 7% the running time of CBC decryption and CTR encryption, respectively.

CCS CONCEPTS

• **Security and privacy** → **Digital signatures**; **Hash functions and message authentication codes**; • **Computer systems organization** → *Pipeline computing*; *Single instruction, multiple data*;

KEYWORDS

AES-NI; SHA-NI; Vector Instructions; Hash-based Digital Signatures; Data Encryption

ACM Reference Format:
Armando Faz-Hernández, Julio López, and Ana Karina D. S. de Oliveira. 2018. SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures. In *APKC'18: 5th ACM ASIA Public-Key Cryptography Workshop , June 4, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3197507.3197511

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APKC'18, June 4, 2018, Incheon, Republic of Korea
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5756-2/18/06…$15.00
https://doi.org/10.1145/3197507.3197511

1 INTRODUCTION

The omnipresence of cryptographic services has influenced modern processor's designs. One proof of that is the support given to the Advanced Encryption Standard (AES) [29] employing extensions to the instruction set architecture known as the AES New Instructions (AES-NI) [15]. Other examples in the same vein are the CRC32 instructions, which aid on error-detection codes, and the carryless multiplier (`CLMUL`), which is used to accelerate the AES-GCM authenticated encryption algorithm [16]. All of these extensions enhance the performance of cryptographic implementations.

Latest processors have added instructions that support cryptographic hash functions. In 2013, Intel [21] announced the specification of the SHA New Instructions (SHA-NI), a set of new instructions for speeding up the execution of SHA1 and SHA2 hash functions [31]. However, SHA-NI-ready processors were available several years later. In 2016, Intel released the Goldmont microarchitecture which targets low power-consumption devices; and in 2017 AMD introduced Zen, a micro-architecture that supports both AES-NI and SHA-NI.

The Single Instruction Multiple Data (SIMD) parallel processing is an optimization strategy increasingly used to develop efficient software implementations. The Advanced Vector eXtensions 2 (AVX2) [26] are SIMD instructions that operate over 256-bit vector registers. The latest generations of Intel processors, such as Haswell, Broadwell, Skylake, and Kaby Lake, have support for AVX2; meanwhile, Zen is the first AMD's architecture that implements AVX2.

The main contribution of this work is to provide a performance benchmark report of cryptographic algorithms accelerated through SHA-NI and AES-NI. We focus on mainstream processors, like the ones found in laptops and desktops, and also those used in cloud computing environments. To that end, we revisited known optimization techniques targeting the capabilities (and limitations) of four micro-architectures: Haswell (an Intel Core i7-4770 processor), Skylake (an Intel Core i7-6700K processor), Kaby Lake (an Intel Core i5-7400 processor), and Zen (an AMD Ryzen 7 1800X processor).

As part of our contributions, we improve the performance of *multiple-message hashing*, which refers to the task of hashing several messages of the same length. We followed two approaches. First, we develop a pipelined implementation using SHA-NI that saves around 18-21% of the running time. Second, we review SIMD vectorization techniques and report timings for the parallel hashing of four and eight messages.

In public-key cryptography, multiple-message hashing serves as a building block on the construction of hash-based digital signatures, such as XMSS [8] and XMSS^{MT} [24]. These signature schemes are strong candidates to be part of a new cryptographic portfolio of

Algorithm 1 Definition of the Update SHA-256 function.

Domain Parameters: The functions σ_o, σ_1, Σ_o, Σ_1, Ch, Maj, and the values $k_0, \ldots, k_{63}$ are defined in Appendix A.
Input: S is the current state, and M is a 512-bit message block.
Output: $\overline{S} = \texttt{Update}(S, M)$.

Message Schedule Phase
1: $(w_0, \ldots, w_{15}) \leftarrow M$ //Split block into sixteen 32-bit words.
2: **for** $t \leftarrow 16$ **to** 63 **do**
3: $w_t \leftarrow \sigma_0(w_{t-15}) + \sigma_1(w_{t-2}) + w_{t-7} + w_{t-16}$
4: **end for**
Update State Phase
5: $(a, b, c, d, e, f, g, h) \leftarrow S$ //Split state into eight 32-bit words.
6: **for** $i \leftarrow 0$ **to** 63 **do**
7: $t_1 \leftarrow h + \Sigma_1(e) + \mathrm{Ch}(e, f, g) + k_i + w_i$
8: $t_2 \leftarrow \Sigma_0(a) + \mathrm{Maj}(a, b, c)$
9: $h \leftarrow g,\ g \leftarrow f,\ f \leftarrow e,\ e \leftarrow d + t_1$
10: $d \leftarrow c,\ c \leftarrow b,\ b \leftarrow a,\ a \leftarrow t_1 + t_2$
11: **end for**
12: $(\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, \overline{f}, \overline{g}, \overline{h}) \leftarrow S$ //Split state into eight 32-bit words.
13: **return** $\overline{S} \leftarrow (a + \overline{a}, b + \overline{b}, c + \overline{c}, d + \overline{d}, e + \overline{e}, f + \overline{f}, g + \overline{g}, h + \overline{h})$

quantum-resistant algorithms [23]. For this reason, we evaluate the impact on the performance of these signature schemes carried by optimizations of multiple-message hashing.

Finally, we revisited software implementation techniques for the AES modes of operation. As a result, we show optimizations for Zen that reduce the running time of AES-CBC decryption and AES-CTR encryption by 12% and 9%, respectively. As a side result, we also improve the implementation of AEGIS, an AES-based algorithm classified in the final round of the authenticated-encryption competition CAESAR [4], reducing by 11% its running time.

All the source codes derived from this research work are released under a permissive software license and can be retrieved from [http://github.com/armfazh/FLO-shani-aesni].

2 THE SHA-256 HASH FUNCTION

In this section we briefly describe the SHA-256 algorithm and its implementation considering several application scenarios. Following the SHA specification [31], Appendix A contains explicit definitions of some auxiliary functions and parameters of the SHA-256 algorithm.

The SHA-256 algorithm keeps track of a *state* of 256 bits. This state is initialized with constant values and is denoted by S_0 (see Equation (18) in Appendix A). The input message M is padded and then split into n 512-bit blocks. Each of these blocks is used to generate a new state according to the following recurrence: $S_j = \texttt{Update}(S_{j-1}, M_j)$ for $j = 1$ to n. Thus, the SHA-256 algorithm defines the hash of M as SHA-256$(M) = S_n$. The Update function consists of two phases: the *message schedule* phase and the *update state* phase, both presented in Algorithm 1.

2.1 Implementing SHA-256 using SHA-NI

The SHA-NI instruction set offers native support for the most critical operations of SHA1 and SHA-256 hash functions. In particular,

Algorithm 2 The Update SHA-256 function using the SHA-NI set.

Input: S is the current state, and M is a 512-bit message block.
Output: $\overline{S} = \texttt{Update}(S, M)$.

Message Schedule Phase
1: Load M into 128-bit vector registers: W_0, W_4, W_8, and W_{12}.
2: **for** $i \leftarrow 0$ **to** 11 **do**
3: $X \leftarrow \texttt{SHA256MSG1}\ (W_{4i},\ W_{4i+4})$
4: $W_{4i+9} \leftarrow \texttt{PALIGNR}\ (W_{4i+12},\ W_{4i+8},\ 4)$
5: $Y \leftarrow \texttt{PADDD}\ (X,\ W_{4i+9})$
6: $W_{4i+16} = \texttt{SHA256MSG2}\ (Y,\ W_{4i+12})$
7: **end for**
Update State Phase
8: $(a, b, c, d, e, f, g, h) \leftarrow S,\ A = \left[a, b, e, f\right],$ and $C = \left[c, d, g, h\right]$.
9: **for** $i \leftarrow 0$ **to** 15 **do**
10: $X \leftarrow \texttt{PADDD}\ (W_{4i},\ K_{4i})$
11: $Y \leftarrow \texttt{PSRLDQ}\ (X,\ 8)$
12: $C \leftarrow \texttt{SHA256RNDS2}\ (C,\ A,\ X)$
13: $A \leftarrow \texttt{SHA256RNDS2}\ (A,\ C,\ Y)$
14: **end for**
15: $(\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, \overline{f}, \overline{g}, \overline{h}) \leftarrow S\ ,\ \overline{A} = \left[\overline{a}, \overline{b}, \overline{e}, \overline{f}\right],$ and $\overline{C} = \left[\overline{c}, \overline{d}, \overline{g}, \overline{h}\right]$
16: $\left[a, b, e, f\right] \leftarrow \texttt{PADDD}\left(A,\ \overline{A}\right)$
17: $\left[c, d, g, h\right] \leftarrow \texttt{PADDD}\left(C,\ \overline{C}\right)$
18: **return** $\overline{S} \leftarrow (a, b, c, d, e, f, g, h)$

the instructions SHA256MSG1, SHA256MSG2, and SHA256RNDS2 aid on the calculation of the Update function of the SHA-256 algorithm.

In this section, we describe the implementation of the Update function using the SHA-NI set (see Algorithm 2). First of all, it is required that the message block (the values $w_0, \ldots, w_{15}$) be stored into four 128-bit vector registers W_0, W_4, W_8, and W_{12} as follows $W_i = [w_i, w_{i+1}, w_{i+2}, w_{i+3}]$. After that, the SHA256MSG1 and SHA256MSG2 instructions will help on the message schedule phase for computing the values $w_{16}, \ldots, w_{63}$.

To have a better understanding of the design rationale of the SHA-NI extensions, we exemplify how to calculate W_{16}, i.e., the first iteration of the for-loop in lines 2-7 of Algorithm 2. First, the SHA256MSG1 instruction updates the vector register X with four 32-bit words calculated as follows:

$$\begin{aligned} X &\leftarrow \texttt{SHA256MSG1}\ (W_0, W_4) \\ &= \texttt{SHA256MSG1}\left(\left[w_0, w_1, w_2, w_3\right], \left[w_4, w_5, w_6, w_7\right]\right) \\ &= \left[\sigma_0(w_1) + w_0,\ \sigma_0(w_2) + w_1,\ \sigma_0(w_3) + w_2,\ \sigma_0(w_4) + w_3\right]. \end{aligned} \tag{1}$$

In the next line, the PALIGNR instruction obtains W_9 from a 32-bit shift applied to the concatenation of W_{12} and W_8:

$$\begin{aligned} W_9 &\leftarrow \texttt{PALIGNR}\ (W_{12}, W_8, 4) \\ &= \texttt{PALIGNR}\left(\left[w_{12}, w_{13}, w_{14}, w_{15}\right], \left[w_8, w_9, w_{10}, w_{11}\right], 4\right) \\ &= \left[w_9, w_{10}, w_{11}, w_{12}\right]. \end{aligned} \tag{2}$$

After that, the vector Y will store the word-wise addition of the vector registers X and W_9:

$$\begin{aligned} Y &\leftarrow \texttt{PADDD}(X, W_9) \\ &= \left[\sigma_0(w_1) + w_0 + w_9,\ \sigma_0(w_2) + w_1 + w_{10},\right. \\ &\qquad \left.\sigma_0(w_3) + w_2 + w_{11},\ \sigma_0(w_4) + w_3 + w_{12}\right]. \end{aligned} \tag{3}$$

Finally, the SHA256MSG2 instruction produces W_{16} from Y and W_{12}:

$$\begin{aligned} W_{16} &\leftarrow \texttt{SHA256MSG2}\,(Y, W_{12}) = \left[w_{16},\ w_{17},\ w_{18},\ w_{19}\right] \\ &= \left[\sigma_0(w_1) + \sigma_1(w_{14}) + w_0 + w_9,\right. \\ &\quad \sigma_0(w_2) + \sigma_1(w_{15}) + w_1 + w_{10}, \\ &\quad \sigma_0(w_3) + \sigma_1(w_{16}) + w_2 + w_{11}, \\ &\quad \left.\sigma_0(w_4) + \sigma_1(w_{17}) + w_3 + w_{12}\right]. \end{aligned} \tag{4}$$

The remainder values $W_{20}, \ldots, W_{60}$ are calculated applying the same strategy. To calculate W_{4i}, $i = 4, \ldots, 15$, depends on $W_{4(i-1)}$, $W_{4(i-2)}$, $W_{4(i-3)}$, and $W_{4(i-4)}$; from which the latter can be used to store the new value allowing to reuse the vector register.

In the update state phase, the SHA256RNDS2 instruction assumes that the state, say $S = (a, b, c, d, e, f, g, h)$, is stored into two 128-bit vector registers as follows: $A = \left[a,\ b,\ e,\ f\right]$ and $C = \left[c,\ d,\ g,\ h\right]$. The reasoning behind this representation relies on the following observation: after processing two iterations of the second for-loop of Algorithm 2, some words of the state remain unmodified. Hence, let A_i and C_i be the values of the state at the i-th iteration of the for-loop (lines 9-14 of Algorithm 2), then $C_{i+2} = A_i$ for $i \geq 0$. Figure 1 illustrates this property.

Based on this property, the SHA256RNDS2 instruction performs two iterations of the state update phase as follows:

$$C_{i+2} = A_i, \quad A_{i+2} = \texttt{SHA256RNDS2}\,(C_i, A_i, X) \tag{5}$$

where X is a vector register containing $\left[w_i + k_i, w_{i+1} + k_{i+1}, \varnothing, \varnothing\right]$, k_i and k_{i+1} are constant values defined in Equation (19) (see Appendix A), and $\varnothing$ is an unused value. Four iterations can be computed applying once again the same property; thus we have

$$C_{i+4} = A_{i+2}, \text{ and } A_{i+4} = \texttt{SHA256RNDS2}\,(C_{i+2}, A_{i+2}, Y), \tag{6}$$

which is equivalent to

$$\begin{aligned} C_{i+4} &= \texttt{SHA256RNDS2}\,(C_i, A_i, X), \text{ and} \\ A_{i+4} &= \texttt{SHA256RNDS2}\,(A_i, C_{i+4}, Y) \end{aligned} \tag{7}$$

such that Y is a register containing $\left[w_{i+2} + k_{i+2}, w_{i+3} + k_{i+3}, \varnothing, \varnothing\right]$. The registers X and Y can be computed as

$$\begin{aligned} X &\leftarrow \texttt{PADDD}\,(K_{4i},\ W_{4i}) \\ &= \left[w_i + k_i,\ w_{i+1} + k_{i+1},\ w_{i+2} + k_{i+2},\ w_{i+3} + k_{i+3}\right] \\ Y &\leftarrow \texttt{PSRLDQ}\,(X,\ 8) \\ &= \left[w_{i+2} + k_{i+2}, w_{i+3} + k_{i+3}, 0, 0\right] \end{aligned} \tag{8}$$

where $K_{4i} = \left[k_{4i},\ k_{4i+1},\ k_{4i+2},\ k_{4i+3}\right]$. As Algorithm 2 shows, this execution pattern is repeated sixteen times to perform the 64 iterations of the update state phase.

2.2 Performance Impact of SHA-NI

In this section, we contrast the performance of several implementations of SHA-256 such as the sphlib library[1], the OpenSSL library[2], and the implementation using SHA-NI instructions.

In the benchmark program, we measured the number of clock cycles that takes to hash a message; then from these measurements, we calculate the cycles-per-byte (cpb) ratio, which is conventionally used as a metric of performance. Figure 2 shows the performance behavior of three implementations on the Zen processor.

[1]The sphlib library by Thomas Pornin was taken from SUPERCOP [6].
[2]We measured the 64-bit implementation of SHA-256 provided by OpenSSL, more details about the optimizations of such implementation can be found at [17, 20].

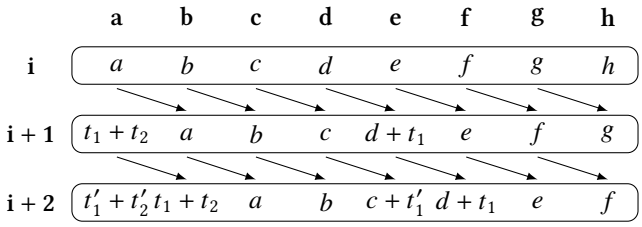

Figure 1: Every two consecutive iterations of the update state phase, it holds that the values (c, d, g, h) at the $(i+2)$-th iteration are exactly the values (a, b, e, f) of the i-th iteration. The values (a, b, e, f) of the $(i+2)$-th iteration are calculated using the SHA256RNDS2 instruction.

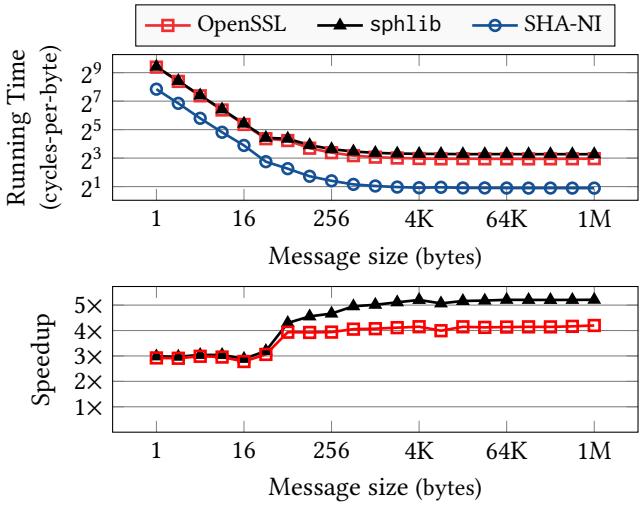

Figure 2: Performance of SHA-256 on the Zen processor. On top, it is shown the cycles-per-byte taken for hashing a message. On the bottom, it is shown the speedup factor yielded by using the SHA-NI set.

2.2.1 Analysis. For messages larger than 256 bytes, the sphlib library takes around 9.6 cpb, whereas, the OpenSSL library offers a better performance taking 7.7 cpb, as is shown on the top of Fig. 2. On the other hand, the SHA-NI implementation takes 1.8 cpb; this is approximately 5.1× faster than sphlib and is 4.2× faster than OpenSSL. The speedup factor yielded by using SHA-NI is plotted on the bottom of Fig. 2. We want to remark that for short-length messages the improvement on the performance is also significant, since it achieves a 3.0× factor of speedup. These performance numbers show the relevance of including specialized instruction sets for supporting cryptographic algorithms.

3 MULTIPLE-MESSAGE HASHING

This task is also known in the literature as multi-buffer hashing, simultaneous hashing, or n-way hashing [1, 13, 17]. This workload

can be efficiently performed using parallel computing techniques, such as the multi-core processing, the SIMD vectorization, and the pipelining instruction scheduling. In this section, we focus on the latter two approaches aiming to improve the performance of SHA-256 in the multiple-message scenario.

3.1 SIMD Vectorization of SHA-256

Vectorization refers to the support given by the processor to the parallel computing paradigm known as Single Instruction Multiple Data (SIMD). Since 2000, commodity processors have included extensions to the instruction set architecture enabling vectorization. One of the first instruction sets is the Streaming SIMD Extensions (SSE) [25], which is a vector unit that operates over 128-bit registers; thus, it is possible to compute four 32-bit operations simultaneously. Later, SSE was improved by the Advanced Vector eXtensions (AVX/AVX2) [26], which is an instruction set that operates over a bank of sixteen 256-bit registers. More recently, Intel launched the Core-X processors, which have support for the AVX512 [27] instruction set and contains a bank of 32 512-bit registers.

To hash n messages of the same length using vector instructions, one can modify Algorithm 1 as follows: first, replace each 32-bit word by a vector register; thus, a set of eight vector registers $A, \ldots, H$ will represent the state, where $A = [a_1, \ldots, a_n]$ is a vector register containing the word a of each message, and the same rationale applies to the rest of the state. Since now all variables are vectors, all the binary (scalar) operations must be replaced by the corresponding vector instructions, which execute simultaneous operations in each lane of a vector register. At the end of the algorithm, the digest of the i-th message must be recovered concatenating the i-th lane of each vector register $A, \ldots, H$. Refer to Appendix B for a vectorized implementation of Algorithm 1.

In the literature, there exist several works that used SIMD instructions for implementing multiple-message hashing. For instance, Gueron [17] showed the n-SMS technique that parallelizes the message schedule phase leading to a faster single message hashing. Following the work of Aciicmez [1], Gueron and Krasnov [18] reported an implementation that uses the SSE unit to compute four SHA-256 digests simultaneously; as a result, their implementation runs 2.2× faster than the n-SMS single-message implementation of Gueron [17]. They also extended the parallelization to 256-bit registers, however, although AVX2 was not available at the time their work was published, their estimations accurately matched the performance exhibited by AVX2-ready processors available nowadays. Recently, Intel [12] contributed to the OpenSSL project with optimized code[3] that computes either four digests using SSE or eight digests using AVX2 instructions.

We want to determine the impact on the running time when vectorization is applied to the multiple-message hashing scenario. To do that, we conducted a performance benchmark of the vectorized implementation provided by the OpenSSL library. Figure 3 shows the result of our measurements.

3.1.1 Analysis. On Kaby Lake, to compute four hashes simultaneously is 2.35× faster than four consecutive invocations to the single-message implementation. Moreover, the AVX2 implementation is 4.5× faster than `sphlib` for computing eight hashes in a row. The use of vector instructions on Kaby Lake shows a noticeable improvement in the performance of multiple-message hashing. However, the story is different on Zen.

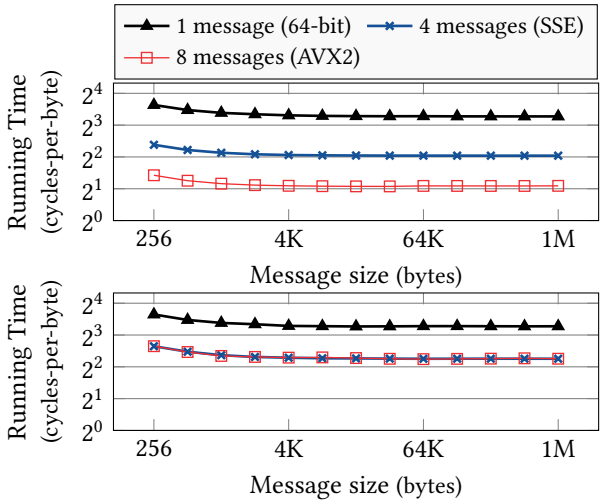

Figure 3: Performance of multiple-message hashing on Kaby Lake (top) and on Zen (bottom). The baseline single-message implementation is the `sphlib` library [6].

The graphs in Figure 3 show that Zen offers a similar performance than Kaby Lake for single-message hashing. The SSE implementation, which computes four hashes simultaneously, renders a better performance on Kaby Lake. On Zen, the performance of the AVX2 implementation shows the same performance as the one exhibited by the SSE implementation, i.e., no benefits were observed by running AVX2 code on the Zen micro-architecture.

This performance downgrade on Zen is because the latency of AVX2 instructions is twice slower than the latency of the SSE instructions. The micro-architectural design of Zen *emulates* a 256-bit vector instruction splitting the workload into two parts, and then it executes them in a 128-bit vector unit sequentially. Therefore, the expected performance of an AVX2 code is reduced by half on Zen.

3.2 Pipelining SHA-NI Instructions

This section presents optimization strategies that rely on an efficient instruction scheduling to leverage the capabilities of the processor's pipeline. First, we briefly describe how the processor's pipeline operates, and detail about the pipelined implementation of the SHA-256 algorithm.

In former computer architectures and before the introduction of Reduced Instruction Set Computer (RISC) processor's design, instructions do not start their execution until the previous instructions are executed completely. In this design, the processor stalls whenever executing high-latency instructions. To remedy this issue, RISC introduced the *pipeline* execution model, which is a hardware optimization for increasing the throughput of instruction execution. In this model, once an instruction has been issued, then it is processed by several (and simpler) stages until it is completely

[3] The OpenSSL library (v.1.0.2) contains the `sha256_multi_block` function, which is located in the file http://github.com/openssl/openssl/blob/OpenSSL_1_0_2-stable/crypto/sha/asm/sha256-mb-x86_64.pl .

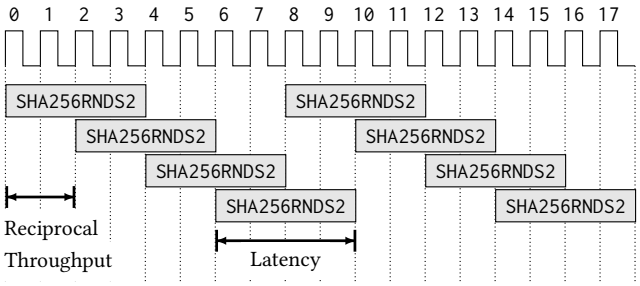

Figure 4: Pipelined execution of `SHA256RNDS2` instructions.

executed. Inside the pipeline, the execution of instructions is overlapped; this means that at a given time, the execution of the stage t of an instruction happens at the same time as the execution of the stage $t+1$ of the previous instruction, and also at the same time as the execution of the stage $t-1$ of the next instruction.

In addition to the latency, there exist other metrics to determine the performance of instructions on a RISC architecture. According to Fog's definition [11], the throughput of an instruction is "the maximum number of instructions of the same kind that are executed per clock cycle", and the reciprocal throughput refers to "the number of cycles to wait until an execution unit starts processing an instruction of the same type". There is a lack of information about these metrics on the Zen documentation; however, accurate timings can be found using experimental measurements. The following table shows the timings measured by Fog for the SHA-NI instructions on Zen:

	`SHA256MSG1`	`SHA256MSG1`	`SHA256RNDS2`
Latency	2	3	4
Recip. Throughput	0.5	2	2

In Figure 4, we show a timeline of the pipelined execution of a series of `SHA256RNDS2` instructions. The `SHA256RNDS2` instruction takes four clock cycles to be completed; however, its reciprocal throughput is only two cycles, this means that once a `SHA256RNDS2` instruction is issued to the execution unit, a second `SHA256RNDS2` instruction can be issued two clock cycles after the first one. Therefore, by issuing two `SHA256RNDS2` instructions consecutively, the processor will take only six clock cycles to compute them instead of taking eight clock cycles.

Observe that to achieve a pipelined execution; these instructions must have no data dependencies; otherwise, the processor will wait until the data dependency be resolved. After a dependence analysis of Algorithm 2, it can be noted that in the message schedule phase, the `SHA256MSG2` instruction takes as an input the value produced by the `SHA256MSG1` instruction. Another data dependency is found in the update state phase, where both `SHA256RNDS2` instructions dependent one to the other. Hence, these data dependencies limit the use of the pipelining technique for single-message hashing. However, the hashing of multiple messages is a suitable scenario that leverages the capabilities of the processor's pipeline.

The central idea of our pipelined implementation resembles the work of Gueron [14], who implemented the AES-CTR encryption algorithm using AES-NI instructions. In the case of SHA-256, at

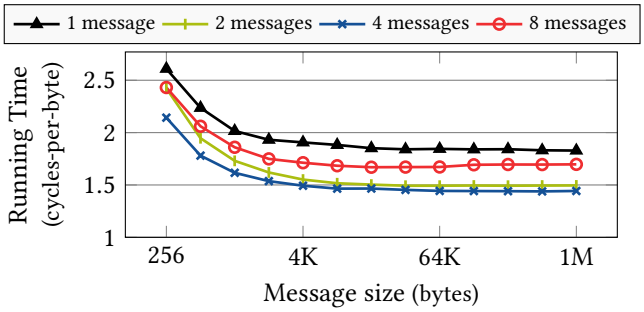

Figure 5: Performance of SHA-256 multiple-message hashing measured on Zen. The single-message implementation refers to the SHA-NI implementation (cf. Section 2.1).

every two clock cycles a `SHA256RNDS2` instruction is issued to the pipeline, such that each instruction operates over a state of a different message; thus, consecutive instructions do not have data dependencies at all, and as a consequence, their execution can be pipelined. Let k be the number of messages to be hashed; we developed pipelined implementations of SHA-256 multiple-message hashing for $k \in \{2, 4, 8\}$. Appendix B shows the pipelined implementation of Algorithm 2 for $k = 2$. The results of the performance benchmark are shown in Figure 5.

3.2.1 Analysis. To hash two messages, it is more convenient to use the pipelined implementation ($k = 2$), which is 18% faster than performing two consecutive hashes using the single-message SHA-NI implementation. Similarly, for $k = 4$ a reduction of a 21% of the running time is obtained. However, note that for $k = 8$ the performance downgrades, which can be explained because hashing eight messages requires a larger space to store all the states causing that vector registers be spilled to memory more often. These results show that a significant improvement for multiple-message hashing is achieved using an efficient instruction scheduling of SHA-NI instructions.

4 HASH-BASED DIGITAL SIGNATURES

We present a performance analysis of two hash-based signature schemes that are currently considered as a work in progress towards standardization by the Internet Engineering Task Force (IETF) group [23].

The first scheme is the eXtended Merkle Signature Scheme (XMSS) [8], which uses a Merkle tree (a binary hash-tree) in which the leaves store a public key of a one-time signature scheme, such as WOTS$^+$ [22]; and every internal node stores the hash of the values of its child nodes. Thus, the XMSS public key is the root of the Merkle tree. Let h be the height of the tree, then XMSS can sign at most 2^h messages using 2^h different key pairs, where the one-time private keys are derived from a secret seed using a pseudo-random number generator.

The second scheme is the Multi-Tree XMSS (XMSS^{MT}) [24], which is an extension of XMSS that provides a larger number of signatures. Assuming that $d|h$, XMSS^{MT} can generate at most 2^h signatures using a d-height hypertree such that each of its nodes is an h/d-height XMSS tree.

4.1 Implementation of XMSS and XMSS^{MT}

In both schemes, the calculation of Merkle trees is the performance-critical operation, since a large number of hash calculations is required. Note that the nodes at the same height can calculate their hash values without dependency between them; hence, this is a suitable scenario for applying the optimized implementations of multiple-message hashing presented above.

To implement the signature schemes we use as a building block the following software implementations of the SHA-256 algorithm:

- The `sphlib` [6] implementation that uses 64-bit instructions.
- The sequential SHA-NI implementation from Section 2.1.
- Vectorized implementations using 128- and 256-bit instructions from [9].
- The pipelined SHA-NI implementation from Section 3.2.

Optimal parameters for these schemes are provided in the IEFT's draft [23]. For example, to achieve a 128-bit post-quantum security level it is recommended to use a hash function with an output of $n = 32$ bytes, such as SHA-256 or SHAKE [32]. For XMSS, it is suggested to set $h \in \{10, 16, 20\}$; and for XMSS^{MT} setting $(h, d) \in \{(20, 2), (20, 4), (40, 2), (40, 4), (40, 8), (60, 3), (60, 6), (60, 12)\}$. From these parameters, we selected $h = 20$ for XMSS and $h = 60, d = 6$ for XMSS^{MT} since they allow the largest number of signatures. We report in Table 1 the performance measurements.

4.1.1 Analysis of the Zen Platform. Taking as a baseline the `sphlib` implementation, it can be noted that multiple-message hashing has a higher impact on the key generation and the signing operation and a lesser impact on the verification procedure. Note that the running time of the key generation and the signing operation can be reduced by almost half using SSE vector instructions, whereas the AVX2 implementation renders a poor performance on Zen; this was already expected from the analysis presented in Section 3.1.1.

It can be observed that for XMSS the SHA-NI implementation yields a speedup factor between 3.5× to 4× in contrast to the 64-bit `sphlib` implementation. Moreover, extra savings were achieved by using the pipelined SHA-NI implementation, which increased the speedup factor to 4.3× and 4.6× for XMSS and XMSS^{MT}, respectively, improving a 10% the key generation, and a 7% the signature operation.

4.1.2 Analysis of the Kaby Lake Processor. To have a better panorama of the performance of hash-based signatures, we reproduce the experiments using Kaby Lake. Recall that although Kaby Lake does not support the SHA-NI instructions, it contains a faster vector unit. Figure 6 shows the running time to generate an XMSS signature ($h = 20$) measured on Zen and Kaby Lake.

First of all, it can be observed that Zen delivers better performance for signing using the sequential `sphlib` implementation. However, the vectorized implementations offer a significant acceleration when running on Kaby Lake, but not in Zen. For example, by using AVX2, an XMSS signature can be computed 4× faster than the sequential implementation. On the other hand, Zen renders a similar performance using the sequential SHA-NI implementation. However, the pipelined SHA-NI implementation (from Section 3.2) offers the fastest timings. These results show the relevance of revisiting optimization techniques for the SHA-NI set.

Table 1: Performance comparison of hash-based signature schemes on Zen. For each signature operation, the first column lists the latency of the operation and the second column shows the acceleration factor (AF) with respect to the 64-bit `sphlib` implementation.

(a) XMSS for $h = 20$.

Impl.	Parallel	Key Gen.		Sign		Verify	
		10^{12} cc	AF	10^6 cc	AF	10^6 cc	AF
sphlib	No	4.50	1.00×	21.56	1.00×	2.16	1.00×
SSE	4-way	2.60	1.72×	12.76	1.68×	1.78	1.21×
AVX2	8-way	3.81	1.18×	19.56	1.10×	3.65	0.59×
SHA-NI	No	1.12	4.01×	5.39	3.99×	0.61	3.51×
SHA-NI	4-Pipelined	1.01	4.46×	5.00	4.30×	0.74	2.89×

(b) XMSS^{MT} for $h = 60$ and $d = 6$.

Impl.	Parallel	Key Gen.		Sign		Verify	
		10^9 cc	AF	10^6 cc	AF	10^6 cc	AF
sphlib	No	51.63	1.00×	46.35	1.00×	27.97	1.00×
SSE	4-way	27.44	1.88×	25.27	1.83×	18.94	1.48×
AVX2	8-way	40.11	1.29×	38.31	1.21×	36.81	0.76×
SHA-NI	No	11.87	4.35×	10.69	4.34×	6.45	4.33×
SHA-NI	4-Pipelined	10.78	4.79×	9.99	4.64×	7.64	3.66×

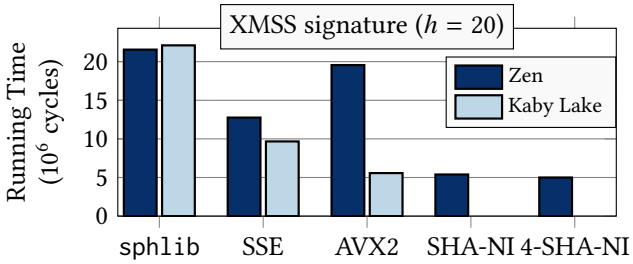

Figure 6: The faster implementation of SHA-256, the better performance achieved for XMSS signature generation.

5 IMPLEMENTATION OF THE AES

The Advanced Encryption Standard (AES) is a family of block ciphers standardized by NIST [29] that encrypts a 128-bit message using a secret key k to produce a 128-bit ciphertext. The algorithm keeps track of a 128-bit state initially containing the message input. Then, the state is updated by the application of the functions AddRoundKey, SubBytes, ShiftRows, and MixColumns to produce the ciphertext finally. The AES algorithm admits three key sizes: 128, 192, and 256 bits; this size defines the number of transformation rounds that the state is processed. A description of the AES encryption algorithm is shown in Algorithm 3.

In 2010, Intel released the AES-NI set [15], which contains six instructions dedicated to calculating parts of the AES algorithm. In particular, the `AESENC` and `AESENCLAST` instructions encapsulate the operations performed in each round. Thus, Algorithm 3 can be implemented replacing the lines 4-7 by an `AESENC` instruction and

Algorithm 3 The AES block cipher encryption algorithm.

Input: M is a 128-bit message, and k is a secret key such that $(|k|, n_r) \in \{(128, 10), (192, 12), (256, 14)\}$.
Output: C is a 128-bit ciphertext such that $C = \text{AES}_k(M)$.

1: $(K_0, \ldots, K_{n_r}) \leftarrow \texttt{KeyExpansion}(k)$
2: $S \leftarrow \texttt{AddRoundKey}(M, K_0)$
3: **for** $i \leftarrow 1$ **to** $n_r - 1$ **do**
4: $\quad S \leftarrow \texttt{SubBytes}(S)$
5: $\quad S \leftarrow \texttt{ShiftRows}(S)$
6: $\quad S \leftarrow \texttt{MixColumns}(S)$
7: $\quad S \leftarrow \texttt{AddRoundKey}(S, K_i)$ } $\texttt{AESENC}(S, K_i)$
8: **end for**
9: $S \leftarrow \texttt{SubBytes}(S)$
10: $S \leftarrow \texttt{ShiftRows}(S)$ } $\texttt{AESENCLAST}(S, K_{n_r})$
11: $C \leftarrow \texttt{AddRoundKey}(S, K_{n_r})$
12: **return** C

the lines 9-11 by an AESENCLAST instruction. The resultant AES-NI implementation renders a faster performance since a large part of its execution is performed by hardware.

A *mode of operation* is an algorithm to encrypt arbitrary large messages using a block cipher. The most used modes of operation for AES are the Cipher Block Chaining (CBC) and the Counter mode (CTR) [30]. In the CBC mode, encryption is a sequential process; whereas decryption exhibits a large degree of parallelism. Likewise, the CTR mode is an embarrassingly parallel workload.

The Zen micro-architecture offers a new capability for improving the execution of the AES algorithm. It includes two AES execution units, one more unit than Intel processors. This fact motivated us to revisit implementation techniques of the CBC and CTR modes and to show some optimizations that take advantage of these units.

5.1 Multiple-Message Encryption

To determine the impact on performance due to the inclusion of a second AES-NI unit, we perform a benchmark of the multiple-message encryption using the AES-128-CBC mode. In this task, each message is independent of each other; hence, no data dependencies occur at all. Figure 7 shows our measurements.

5.1.1 Analysis. For encrypting a message, the fastest timing was obtained by Kaby Lake (2.33 cpb), whereas Zen is slightly slower taking 2.44 cpb. On the other hand, as the number of messages increases, Zen executes AESENC instructions more efficiently than the other processors; for instance, Zen is 42% faster than Skylake to encrypt $k = 8$ messages. However, although Zen has two units, Kaby Lake reduced such advantage to 33%.

5.2 Pipelining CBC Decryption and CTR Modes

It is well known that the running time of these modes of operation can be accelerated through proper usage of the processor's pipeline. Gueron [14] showed that encrypting a set of $w > 1$ consecutive blocks leads to a better performance. His idea relies on calculating the first round of each block, i.e., a series of w AESENC instructions is issued; after that, the second round of each message is processed; and this strategy is repeated until the last round. Since each AESENC

k	Hsw	Sky	Kaby	Zen
1	4.49	2.71	2.33	2.44
2	2.25	1.37	1.17	1.24
4	1.13	0.69	0.60	0.63
6	0.81	0.65	0.56	0.43
8	0.76	0.63	0.54	0.36

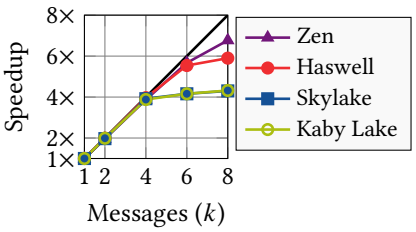

Figure 7: The table entries represent the cpb taken for encrypting k messages of 1 MB using the AES-128-CBC mode.

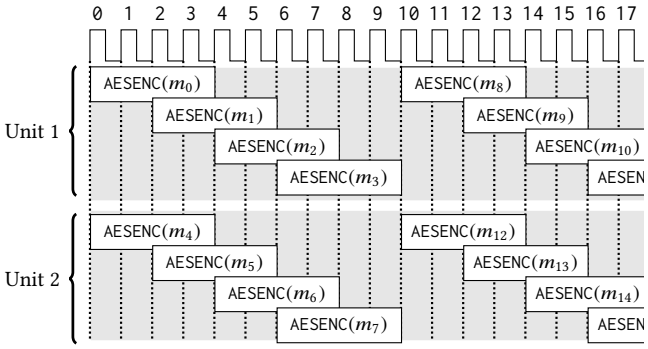

Figure 8: Instruction scheduling of AESENC instructions on Zen. Four AESENC instructions can be issued to each unit.

instruction operates on a different state, no data dependencies will occur, and the processor's pipeline will be able to execute them overlapped.

A detailed analysis of pipelined implementations was presented by Bogdanov et al. [7]. They refer to w as the parameter that expresses the parallelism degree of the instruction, and its optimal value is given as the product of the latency times the throughput of the instruction. Thus, according to the measurements obtained by Fog [11], the optimal value for the AESENC instruction is as follows:

	Haswell	Skylake	Kaby Lake	Zen
Latency	7	4	4	4
Reciprocal throughput	1	1	1	0.5
Parallelism degree (w)	7	4	4	8

Like Kaby Lake, Zen executes the AESENC instruction in four clock cycles; however, since Zen has two AES units, then two independent AESENC instructions can start its execution at the same time, which explains a (fractional) reciprocal throughput of 0.5 cycles. See in Figure 8 the pipelined execution on two units. Note that the optimal value of w is composed of the optimal value of each unit; thus, w_{Zen} = number of units × latency × throughput-per-unit = 8. In Figure 9, we report measurements of several pipelined implementations of AES-CBC decryption and AES-CTR encryption.

5.2.1 Analysis. Both Skylake and Kaby Lake obtain the fastest timing using $w = 2$ for CBC decryption; however, the timings are not improved for larger values of w. Similarly, the CTR encryption reaches its fastest timing using $w = 4$. In these two platforms, no

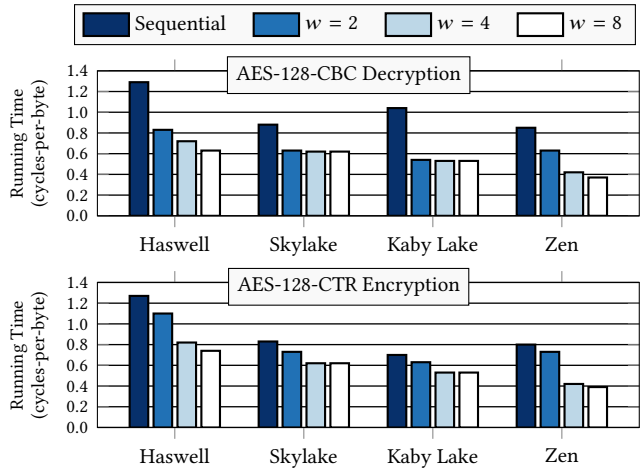

Figure 9: Performance of AES-128 encryption of 1 MB message using CBC decryption and CTR encryption. The w parameter indicates the number of independent `AESENC` instructions issued to the pipeline.

benefits were observed by encrypting $w = 8$ consecutive blocks. However, Haswell saves additional clock cycles setting $w = 8$. For example, the running time of CBC decryption decreases from 0.72 to 0.63 cpb (12.5% faster) and the running time of CTR encryption reduces from 0.82 to 0.74 cpb (9.75% faster).

On the other hand, significant savings were observed for Zen. The $w = 4$ pipelined implementation is twice as fast as using the sequential implementation for both modes of operation leading to 0.42 cpb for both CBC decryption and CTR encryption. Moreover, we observed extra savings by using the $w = 8$ pipelined implementation, which reduces to 0.37 cpb (11.9% faster) the running time of CBC decryption and to 0.39 cpb (7.1% faster) the running time of CTR encryption.

5.3 Improving the Implementation of AEGIS

AEGIS is an authenticated encryption algorithm proposed by Wu and Prenel [35] that is currently classified in the final round of the CAESAR competition [4]. This cipher uses the `AESENC` instruction as a building block to update an 80-byte state. Thus, given a 128-bit message block m_i and the state $S_i = \{a_i, b_i, c_i, d_i, e_i\}$, the `Update` function is defined as: $S_{i+1} = \texttt{Update}(S_i, m_i)$ such that:

$$\begin{aligned} a_{i+1} &= \texttt{AESENC}(e_i, a_i \oplus m_i)\,, & b_{i+1} &= \texttt{AESENC}(a_i, b_i)\,, \\ c_{i+1} &= \texttt{AESENC}(b_i, c_i)\,, & d_{i+1} &= \texttt{AESENC}(c_i, d_i)\,, \\ e_{i+1} &= \texttt{AESENC}(d_i, e_i)\,. \end{aligned} \tag{9}$$

As it can be noted, the five calls to `AESENC` instruction are pairwise independent, which allows benefiting from the processor's pipeline.

Based on the AEGIS implementation available in SUPERCOP [6], we modify it by grouping blocks of four `AESENC` instructions corresponding to the update of b, c, d, and e blocks. The invocation of the last `AESENC` instruction is slightly modified; instead of passing as input $a_i \oplus m_i$, we pass a memory address that points to the message block m_i; thus, the a block is computed as

$$a_{i+1} = \texttt{AESENC}(e_i, a_i \oplus m_i) = \texttt{AESENC}(e_i, m_i) \oplus a_i \tag{10}$$

This equivalence works because the `AddRoundKey` function is an XOR between the state and the round key. This subtle change accelerates its execution since the latency of the memory access is covered by the latency of `AESENC`. The original implementation takes 0.34 cpb in Zen; whereas our optimization leads to 0.30 cpb, which is 11% faster.

6 FINAL REMARKS

Part of this study determined the performance improvements carried by the use of the SHA-NI, the AES-NI, and the SIMD instruction sets available on modern computer architectures.

Regarding SHA-NI extensions, we looked for optimizations that allow efficient use of these instructions. Consequently, we revisited known software optimization techniques, such as vectorization and pipelining, to provide faster implementations of both public-key and symmetric-key cryptographic algorithms.

In the case of public-key cryptography, we optimized the execution of multiple-message hashing, since it is a critical operation of hash-based digital signatures. We evaluated the performance of XMSS and XMSS$^{\text{MT}}$ using several optimization variants. For example, a pipelined SHA-NI implementation yields a speedup factor of 4.3× in the calculation of XMSS signatures and 4.6× in the calculation of XMSS$^{\text{MT}}$ signatures with respect to an optimized 64-bit implementation of SHA-256.

For symmetric-key encryption, we measure the impact on the performance of data encryption by using two AES units. For example, Kaby Lake can decrypt data using the CBC mode at a rate of 0.53 cpb using a $w = 8$ pipelined implementation, whereas Zen is 30% faster taking 0.37 cpb.

We want to highlight that the architectural design of the AVX2 vector unit of Zen affects the performance of AVX2 code negatively. This poor performance was not only observed for hashing but also on AVX2 code supporting the X25519 Diffie-Hellman function [10].

Future works. Note that SIMD parallel implementations can also be extended for using AVX512 instructions [27]; thus, the multiple-message hashing will process sixteen messages simultaneously. Following an analogous analysis that the one performed by Gueron and Krasnov [19]; preliminary evaluation of an implementation of SHA-256 using AVX512 reveals that the number of instructions does not increase significantly, which could lead to performance improvements on the forthcoming AVX512-ready processors.

It would be interesting to reproduce our experiments on ARM processors that support the ARMv8 Cryptographic Extensions [3] and/or the Scalable Vector Extensions [34], however the performance evaluation in this platform turns to be more complex due to the wide variety of processor's implementations.

Finally, the optimizations presented in this work are also applicable to other algorithms, such as the SPHINCS+ [5] hash-based signature scheme, which was recently submitted to the NIST's Post-Quantum call for proposals [33]; and Deoxys [28] and COLM [2], AES-based authenticated encryption algorithms also contending in the CAESAR competition.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments. The authors acknowledge support during the

development of this research from Intel and FAPESP under project "Secure Execution of Cryptographic Algorithms" under Grant No. 14/50704-7.

A DEFINITIONS USED ON SHA-256 ALGORITHM

The SHA-256 algorithm [31] requires some auxiliary functions and constants, which are listed in this section.

Notation: entries of the matrix are given in base 16; the variables x, y, and z denote 32-bit words; the operators $\wedge$, $\oplus$, and $\neg$ denote, respectively, the AND, XOR, and NOT boolean operations; and the symbols $\ll$ and $\gg$ denote, respectively, a left and right 32-bit shift.

$$\mathrm{Rot}(x,n) = (x \gg n) \oplus (x \ll (32-n)) \tag{11}$$

$$\sigma_0(x) = \mathrm{Rot}(x,7) \oplus \mathrm{Rot}(x,18) \oplus (x \gg 3) \tag{12}$$

$$\sigma_1(x) = \mathrm{Rot}(x,17) \oplus \mathrm{Rot}(x,19) \oplus (x \gg 10) \tag{13}$$

$$\Sigma_0(x) = \mathrm{Rot}(x,2) \oplus \mathrm{Rot}(x,13) \oplus \mathrm{Rot}(x,22) \tag{14}$$

$$\Sigma_1(x) = \mathrm{Rot}(x,6) \oplus \mathrm{Rot}(x,11) \oplus \mathrm{Rot}(x,25) \tag{15}$$

$$\mathrm{Ch}(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{16}$$

$$\mathrm{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge y) \tag{17}$$

To initialize the state, use the following values:

$$S_0 = \begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix} = \begin{bmatrix} \texttt{6a09e667} & \texttt{bb67ae85} & \texttt{3c6ef372} & \texttt{a54ff53a} \\ \texttt{510e527f} & \texttt{9b05688c} & \texttt{1f83d9ab} & \texttt{5be0cd19} \end{bmatrix} \tag{18}$$

The following matrix contains 64 constant values known as k_i:

$$\begin{bmatrix} K_0 \\ K_4 \\ \vdots \\ K_{60} \end{bmatrix} = \begin{bmatrix} k_0 & \dots & k_3 \\ \vdots & \ddots & \vdots \\ k_{60} & \dots & k_{63} \end{bmatrix} = \begin{bmatrix} \texttt{428a2f98} & \texttt{71374491} & \texttt{b5c0fbcf} & \texttt{e9b5dba5} \\ \texttt{3956c25b} & \texttt{59f111f1} & \texttt{923f82a4} & \texttt{ab1c5ed5} \\ \texttt{d807aa98} & \texttt{12835b01} & \texttt{243185be} & \texttt{550c7dc3} \\ \texttt{72be5d74} & \texttt{80deb1fe} & \texttt{9bdc06a7} & \texttt{c19bf174} \\ \texttt{e49b69c1} & \texttt{efbe4786} & \texttt{0fc19dc6} & \texttt{240ca1cc} \\ \texttt{2de92c6f} & \texttt{4a7484aa} & \texttt{5cb0a9dc} & \texttt{76f988da} \\ \texttt{983e5152} & \texttt{a831c66d} & \texttt{b00327c8} & \texttt{bf597fc7} \\ \texttt{c6e00bf3} & \texttt{d5a79147} & \texttt{06ca6351} & \texttt{14292967} \\ \texttt{27b70a85} & \texttt{2e1b2138} & \texttt{4d2c6dfc} & \texttt{53380d13} \\ \texttt{650a7354} & \texttt{766a0abb} & \texttt{81c2c92e} & \texttt{92722c85} \\ \texttt{a2bfe8a1} & \texttt{a81a664b} & \texttt{c24b8b70} & \texttt{c76c51a3} \\ \texttt{d192e819} & \texttt{d6990624} & \texttt{f40e3585} & \texttt{106aa070} \\ \texttt{19a4c116} & \texttt{1e376c08} & \texttt{2748774c} & \texttt{34b0bcb5} \\ \texttt{391c0cb3} & \texttt{4ed8aa4a} & \texttt{5b9cca4f} & \texttt{682e6ff3} \\ \texttt{748f82ee} & \texttt{78a5636f} & \texttt{84c87814} & \texttt{8cc70208} \\ \texttt{90befffa} & \texttt{a4506ceb} & \texttt{bef9a3f7} & \texttt{c67178f2} \end{bmatrix} \tag{19}$$

B SOURCE CODE OF THE UPDATE FUNCTION

We show a fragment of C-language code that shows the usage of SSE/AVX2 and SHA-NI instructions on the implementation of the `Update` function of SHA-256. Add the compiler flags `-mssse3` and `-msha` to enable SSE and SHA-NI instruction sets.

```c
/* @author: Armando Faz (2017) - Public Domain */
#include <stdint.h>
#include <immintrin.h>

uint32_t K[64] = {0x428a2f98, 0x71374491, ..., 0xc67178f2};

#define VEC __m128i
#define LOAD(X)      _mm_load_si128((VEC *)X)
#define STORE(X,Y)   _mm_store_si128((VEC *)X,Y)
#define ADD(X,Y)     _mm_add_epi32(X,Y)
#define SHL(X,Y)     _mm_slli_epi32(X,Y)
#define SHR(X,Y)     _mm_slli_epi32(X,Y)
#define HIGH(X)      _mm_srli_si128(X,8)
#define AND(X,Y)     _mm_and_si128(X,Y)
#define ANDN(X,Y)    _mm_andnot_si128(X,Y)
#define OR(X,Y)      _mm_or_si128(X,Y)
#define XOR(X,Y)     _mm_xor_si128(X,Y)
#define BROAD(X)     _mm_set1_epi32(X)
#define ALIGNR(X,Y)  _mm_alignr_epi8(X,Y,4)
#define L2B(X)       _mm_shuffle_epi8(X,_mm_set_epi32( \
  0x0c0d0e0f,0x08090a0b,0x04050607,0x00010203))
#define SHA(X,Y,Z)   _mm_sha256rnds2_epu32(X,Y,Z)
#define MSG1(X,Y)    _mm_sha256msg1_epu32(X,Y)
#define MSG2(X,Y)    _mm_sha256msg2_epu32(X,Y)
#define CVLO(X,Y,Z) _mm_shuffle_epi32(_mm_unpacklo_epi64(X,Y),Z)
#define CVHI(X,Y,Z) _mm_shuffle_epi32(_mm_unpackhi_epi64(X,Y),Z)
#define ROT(X,N)     OR(SHR(X,N),SHR(X,32-N))
#define sigma0(X)    XOR(ROT(X,7),XOR(ROT(X,18),SHR(X,3)))
#define sigma1(X)    XOR(ROT(X,17),XOR(ROT(X,19),SHR(X,10)))
#define Sigma0(X)    XOR(ROT(X,2),XOR(ROT(X,13),ROT(X,22)))
#define Sigma1(X)    XOR(ROT(X,6),XOR(ROT(X,11),ROT(X,25)))
#define Ch(X,Y,Z)    XOR(AND(X,Y),ANDN(X,Z))
#define Maj(X,Y,Z)   XOR(AND(X,Y),XOR(AND(X,Z),AND(Y,Z)))
#define calcWi(W,t) ADD(sigma0(W[t-15]),\
  ADD(sigma1(W[t-2]),ADD(W[t-7],W[t-16])))
#define calcT1(E,F,G,H,Ki,Wi) \
  ADD(H,ADD(Sigma1(E),ADD(Ch(E,F,G),ADD(Ki,Wi))))
#define calcT2(A,B,C) ADD(Sigma0(A),Maj(A,B,C))

/* 1) Vectorized Implementation */
void Update_AVX2(VEC * state, VEC * msg_block) {
  int i=0;  VEC a,b,c,d,e,f,g,h,ki,T1,T2,M[64];
  a = state[0];  b = state[1];  c = state[2];  d = state[3];
  e = state[4];  f = state[5];  g = state[6];  h = state[7];
  for(i=0;i<16;i++) {
    ki = BROAD(K[i]);  M[i] = msg_block[i];
    T1 = calcT1(e,f,g,h,ki,M[i]); T2 = calcT2(a,b,c);
    h=g; g=f; f=e; e=ADD(d,T1); d=c; c=b; b=a; a=ADD(T1,T2);
  }
  for(i=16;i<64;i++) {
    ki = BROAD(K[i]);  M[i] = calcWi(M,i);
    T1 = calcT1(e,f,g,h,ki,M[i]); T2 = calcT2(a,b,c);
    h=g; g=f; f=e; e=ADD(d,T1); d=c; c=b; b=a; a=ADD(T1,T2);
  }
  state[0] = ADD(state[0],a);  state[1] = ADD(state[1],b);
  state[2] = ADD(state[2],c);  state[3] = ADD(state[3],d);
  state[4] = ADD(state[4],e);  state[5] = ADD(state[5],f);
  state[6] = ADD(state[6],g);  state[7] = ADD(state[7],h);
}
/* 2) Pipelined SHA-NI Implementation (k=2) */
void Update_SHANI(uint32_t state0[8], uint8_t * msg0,
                  uint32_t state1[8], uint8_t * msg1) {
  int i=0,j=0,i1=0,i2=0,i3=0;
  VEC X0,Y0,A0,C0,W0[4],X1,Y1,A1,C1,W1[4],Ki;

  X0 = LOAD(state0+0);    X1 = LOAD(state1+0);
  Y0 = LOAD(state0+1);    Y1 = LOAD(state1+1);
  A0 = CVLO(X0,Y0,0x1B);  A1 = CVLO(X1,Y1,0x1B);
  C0 = CVHI(X0,Y0,0x1B);  C1 = CVHI(X1,Y1,0x1B);

  for(i=0; i<4; i++) {
    Ki = LOAD(K+i);
    W0[i] = L2B(LOAD(msg0+i)); W1[i] = L2B(LOAD(msg1+i));
    X0 = ADD(W0[i],Ki);        X1 = ADD(W1[i],Ki);
    Y0 = HIGH(X0);             Y1 = HIGH(X1);
    C0 = SHA(C0,A0,X0);        C1 = SHA(C1,A1,X1);
    A0 = SHA(A0,C0,Y0);        A1 = SHA(A1,C1,Y1);
  }
  for(j=1; j<4; j++) {
    for(i=0, i1=1, i2=2, i3=3; i<4; i++) {
      Ki = LOAD(K+4*j+i);
      X0 = MSG1(W0[i],W0[i1]);     X1 = MSG1(W1[i],W1[i1]);
      Y0 = ALIGNR(W0[i3],W0[i2]);  Y1 = ALIGNR(W1[i3],W1[i2]);
      X0 = ADD(X0,Y0);             X1 = ADD(X1,Y1);
      W0[i] = MSG2(X0,W0[i3]);     W1[i] = MSG2(X1,W1[i3]);
      X0 = ADD(W0[i],Ki);          X1 = ADD(W1[i],Ki);
      Y0 = HIGH(X0);               Y1 = HIGH(X1);
```

```
      C0 = SHA(C0,A0,X0);          C1 = SHA(C1,A1,X1);
      A0 = SHA(A0,C0,Y0);          A1 = SHA(A1,C1,Y1);
      i1 = i2;  i2 = i3;  i3 = i;
    }
  }
  X0 = CVHI(A0,C0,0xB1);        X1 = CVHI(A1,C1,0xB1);
  Y0 = CVLO(A0,C0,0xB1);        Y1 = CVLO(A1,C1,0xB1);
  X0 = ADD(X0,LOAD(state0+0));  X1 = ADD(X1,LOAD(state1+0));
  Y0 = ADD(Y0,LOAD(state0+1));  Y1 = ADD(Y1,LOAD(state1+1));
  STORE(state0+0,X0);           STORE(state1+0,X1);
  STORE(state0+1,Y0);           STORE(state1+1,Y1);
}
```

Listing 1: Implementation of the `Update` function using: 1) SSE vectors; 2) pipelined SHA-NI extensions.

REFERENCES

[1] Onur Aciicmez. 2005. *Fast hashing on Pentium SIMD architecture.* Master's thesis. Oregon State University. http://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/mk61rk723

[2] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. 2013. Parallelizable and Authenticated Online Ciphers. In *Advances in Cryptology - ASIACRYPT 2013*, Kazue Sako and Palash Sarkar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–443. https://doi.org/10.1007/978-3-642-42033-7_22

[3] ARM. 2017. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile.* ARM. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

[4] Daniel J. Bernstein (Ed.). 2013. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness.* Cryptographic competitions. https://competitions.cr.yp.to/caesar-submissions.html

[5] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. 2015. SPHINCS: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 368–397. https://doi.org/10.1007/978-3-662-46800-5_15

[6] Daniel J. Bernstein and Tanja Lange. 2017. eBACS: ECRYPT Benchmarking of Cryptographic Systems. (Dec. 2017). http://bench.cr.yp.to/supercop.html Published: Accessed on 20 December 2017.

[7] Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. 2015. Comb to Pipeline: Fast Software Encryption Revisited. In *Fast Software Encryption: 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, Gregor Leander (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–171. https://doi.org/10.1007/978-3-662-48116-5_8

[8] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. 2011. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In *Post-Quantum Cryptography*, Bo-Yin Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–129. https://doi.org/10.1007/978-3-642-25405-5_8

[9] Ana Karina D. S. de Oliveira and Julio López. 2015. An Efficient Software Implementation of the Hash-Based Signature Scheme MSS and Its Variants. In *Progress in Cryptology – LATINCRYPT 2015*, Kristin Lauter and Francisco Rodríguez-Henríquez (Eds.). Springer International Publishing, Guadalajara, Mexico, 366–383. https://doi.org/10.1007/978-3-319-22174-8_20

[10] Armando Faz-Hernández and Julio López. 2015. Fast Implementation of Curve25519 Using AVX2. In *Progress in Cryptology – LATINCRYPT 2015 (Lecture Notes in Computer Science)*, Kristin Lauter and Francisco Rodríguez-Henríquez (Eds.), Vol. 9230. Springer International Publishing, Guadalajara, Mexico, 329–345. https://doi.org/10.1007/978-3-319-22174-8_18

[11] Agner Fog. 2017. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* Technical University of Denmark. http://www.agner.org/optimize/instruction_tables.pdf.

[12] Vinodh Gopal, Sean Gulley, Wajdi Feghali, Dan Zimmerman, and Ilya Albrekht. 2015. *Improving OpenSSL Performance.* Technical Report. Intel Corporation. https://software.intel.com/en-us/articles/improving-openssl-performance

[13] Vinodh Gopal, Jim Gullford, Wajdi Feghali, Erdinc Ozturk, Gil Wolrich, and Martin Dixon. 2010. *Processing Multiple Buffers in Parallel to Increase Performance on ® Intel Architecture Processors.* Technical Report 324101. Intel Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf

[14] Shay Gueron. 2009. Intel's New AES Instructions for Enhanced Performance and Security. In *Fast Software Encryption: 16th International Workshop, FSE 2009 Leuven, Belgium, February 22-25, 2009 Revised Selected Papers*, Orr Dunkelman (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–66. https://doi.org/10.1007/978-3-642-03317-9_4

[15] Shay Gueron. 2010. *Intel® Advanced Encryption Standard (AES) New Instructions Set.* Technical Report. Intel Corporation. http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[16] Shay Gueron and Michael Kounavis. 2010. Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Inform. Process. Lett.* 110, 14 (2010), 549–553. https://doi.org/10.1016/j.ipl.2010.04.011

[17] Shay Gueron and Vlad Krasnov. 2012. Parallelizing message schedules to accelerate the computations of hash functions. *Journal of Cryptographic Engineering* 2, 4 (01 Nov 2012), 241–253. https://doi.org/10.1007/s13389-012-0037-z

[18] Shay Gueron and Vlad Krasnov. 2012. Simultaneous Hashing of Multiple Messages. *Journal of Information Security* 3, 4 (Oct. 2012), 319–325. https://doi.org/10.4236/jis.2012.34039

[19] S. Gueron and V. Krasnov. 2016. Accelerating Big Integer Arithmetic Using Intel IFMA Extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE, Santa Clara, CA, USA, 32–38. https://doi.org/10.1109/ARITH.2016.22

[20] Jim Guilford, Kirk Yap, and Vinodh Gopal. 2012. *Fast SHA-256 Implementations on Intel ® Architecture Processors.* Technical Report 327457-001. Intel Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf

[21] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Gullford, and Gil Wolrich. 2013. *Intel ® SHA Extensions New Instructions Supporting the Secure Hash Algorithm on Intel ® Architecture Processors.* Technical Report. Intel Corporation. https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf

[22] Andreas Hülsing. 2013. W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In *Progress in Cryptology – AFRICACRYPT 2013*, Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–188. https://doi.org/10.1007/978-3-642-38553-7_10

[23] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. 2018. *XMSS: Extended Hash-Based Signatures.* Internet-Draft draft-irtf-cfrg-xmss-hash-based-signatures-12. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures Work in Progress.

[24] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. 2013. Optimal Parameters for XMSS^{MT} . In *Security Engineering and Intelligence Informatics*, Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–208. https://doi.org/10.1007/978-3-642-40588-4_14

[25] Intel Corporation. 2009. Define SSE2, SSE3 and SSE4. http://www.intel.com/support/processors/sb/CS-030123.htm. (Jan. 2009).

[26] Intel Corporation. 2011. Intel® Advanced Vector Extensions Programming Reference. https://software.intel.com/sites/default/files/m/f/7/c/36945. (June 2011).

[27] Intel Corporation. 2016. *Intel® Architecture Instruction Set Extensions Programming Reference.* Intel Corporation. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf

[28] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. 2014. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In *Advances in Cryptology – ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, Palash Sarkar and Tetsu Iwata (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–288. https://doi.org/10.1007/978-3-662-45608-8_15

[29] National Institute of Standards and Technology. 2001. *Advanced Encryption Standard (AES).* Technical Report FIPS PUB 197. NIST, Gaithersburg, MD, USA. https://doi.org/10.6028/NIST.FIPS.197

[30] National Institute of Standards and Technology. 2001. *Recommendation for Block Cipher Modes of Operation.* Technical Report NIST SP 800-38A. NIST, Gaithersburg, MD, USA. https://doi.org/10.6028/NIST.SP.800-38A

[31] National Institute of Standards and Technology. 2002. *Secure Hash Standard.* Technical Report FIPS PUB 180-2. NIST, Gaithersburg, MD, USA. https://doi.org/10.6028/NIST.FIPS.180-4

[32] National Institute of Standards and Technology. 2015. *FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* Technical Report. Gaithersburg, MD, USA. https://doi.org/10.6028/NIST.FIPS.202

[33] National Institute of Standards and Technology. 2016. *Post-Quantum Cryptography Standardization.* Technical Report. NIST, Gaithersburg, MD, USA. https://www.nist.gov/pqcrypto

[34] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (Mar 2017), 26–39. https://doi.org/10.1109/MM.2017.35

[35] Hongjun Wu and Bart Preneel. 2014. AEGIS: A Fast Authenticated Encryption Algorithm. In *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, Tanja Lange, Kristin Lauter, and Petr Lisoněk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–201. https://doi.org/10.1007/978-3-662-43414-7_10