Improved SIMD Implementation of Poly1305

Sreyosi Bhattacharyya and Palash Sarkar
Applied Statistics Unit, Kolkata
Indian Statistical Institute, Kolkata
203, B.T.Road, Kolkata - 700108, India.
email: {bhattacharyya.sreyosi@gmail.com, palash@isical.ac.in}

July 17, 2019

Abstract

Poly1305 is a polynomial hash function designed by Bernstein in 2005. Presently, it is part of several major platforms including the Transport Layer Security protocol. Vectorised implementation of Poly1305 has been proposed by Goll and Gueron in 2015. We provide some simple algorithmic improvements to the Goll-Gueron vectorisation strategy. Implementation of the modified strategy on modern Intel processors shows marked improvements in speed for short messages.

Keywords: MAC, Poly1305, Horner, 256-bit vectorization, SIMD, Intel Intrinsics.

1 Introduction

Confidentiality and integrity of data flowing through the internet is of paramount importance. The Transport Layer Security (TLS) protocol is the leading security protocol for internet communications. TLS provides a variety of primitives for different cryptographic functionalities. Among the algorithms which are part of TLS is Poly1305 [3] (in combination with ChaCha [1]). Apart from TLS, Poly1305 is part of various other cryptographic libraries: It has been standardised by the IETF and it is part of the NaCl [4] library (in combination with Salsa20 [2]). More concretely, Google uses ChaCha-Poly1305 to secure communication between Chrome browsers on Android and Google websites. The Wikipedia page[1] for Poly1305 provides further details about real world deployment of Poly1305.

Given the widespread use of Poly1305, efficient software implementation of the algorithm is an important issue. Modern processors are moving towards providing vector instructions. These instructions allow single-instruction, multiple-data (SIMD) implementation of a variety of algorithms. The importance of vectorisation in modern processors has been highlighted by Bernstein[2].

The present work considers the issue of SIMD implementation of Poly1305 using vector instructions. To the best of our knowledge, the first such implementation of Poly1305 was done by Goll and Gueron [6]. They showed a way to divide the Poly1305 computation into d independent and parallel computation streams. Concrete implementations were provided in [6] for $d = 4$ and $d = 8$ on modern Intel processors.

Suppose $d = 4$. The top level view of the Goll-Gueron algorithm is as follows. The message is formatted into blocks. The algorithm processes 4 blocks at a time. If the number of blocks in the

[1] `https://en.wikipedia.org/wiki/Poly1305`, accessed on 27th June, 2019.

[2] `https://groups.google.com/a/list.nist.gov/forum/#!searchin/pqc-forum/vectorization\%7Csort:date/pqc-forum/mmsH4k3j_1g/JfzP1EBuBQAJ`, accessed on 27th June, 2019.

message is a multiple of 4, then the algorithm uniformly processes all the blocks. On the other hand, if the number of blocks is not a multiple of 4, then at the end the parallelism breaks down and the tail of the message consisting of one to three blocks have to be processed separately.

We provide a simple idea to improve the Goll-Gueron algorithm. The Poly1305 algorithm essentially computes a polynomial over a finite field whose coefficients are the blocks of the message. Prepending the message with some zero blocks (i.e., blocks corresponding to the zero element of the field), the output of the Poly1305 algorithm remains unchanged. We take advantage of this feature by prepending the message with one to three zero blocks so that overall the number of blocks in the message is a multiple of 4. Then the processing of the blocks can be done 4 at a time in a uniform manner.

The above strategy opens up a further opportunity for improvement. Suppose that the number of blocks in the message is 1 modulo 4. Then 3 zero blocks would need to be prepended. Consider the initial 4-way multiplication. This consists of 3 zero blocks and one message blocks. So, applying a general 4-way multiplication routine in this case leads to many multiplications by zeros. This is wasteful. We describe a new method to perform such an initial multiplication which is much faster than a general 4-way multiplication. Similarly, we extend this to the case where the number of blocks in the message is 2 modulo 4. In the case where the number of blocks is 3 modulo 4, there is no advantage in trying to reduce the number of multiplications using a new initial multiplication algorithm.

We have modified the code accompanying the Goll-Gueron paper to obtain an implementation of the 4-way vectorisation strategy outlined above. For message lengths up to 4000 bytes, this leads to significant speed improvements for messages whose numbers of blocks are not multiples of 4. Comparative speed measurements of the new algorithm and the Goll-Gueron algorithm have been made on the Haswell, Kabylake and Skylake processors. On Haswell, in all cases, speed-ups have been obtained with a maximum speed-up of 31.64% for message lengths up to 1 KB. On Skylake, speed-up has been obtained in 94.97% of cases, same speed in 4.08% of cases and a slowdown in 0.95% of cases; the maximum speed-up obtained is 33.04% for message lengths up to 1 KB. On Kaby Lake, speed-up has been obtained in 96.38% of cases, same speed in 3.29% of cases and a slowdown in 0.33% of cases; the maximum speed-up obtained is 34.35% for message lengths up to 1 KB.

Goll and Gueron [6] also describe a 8-way vectorisation strategy. Our idea of simplifying the parallelism and improving the initial multiplication extends to the 8-way vectorisation. More generally, our algorithmic improvement over the Goll-Gueron strategy applies to all processors which support vector instructions.

2 Description of Poly1305 Hash Function

Let $p = 2^{130} - 5$ and $\mathbb{F}_p$ be the finite field of p elements. The Poly1305 hash function maps a message into an element of $\mathbb{F}_p$. In the original description [3] of Poly1305, the message is a sequence of bytes. The later work [6] considered the message to be a sequence of bits; if the number of bits in the message is a multiple of 8, then the description in [6] coincides with the original description in [3]. Following [6], we provide a description of Poly1305 where a message is a sequence of bits.

Suppose a message M consists of $L \geq 0$ bits. If $L = 0$, define ℓ to be 0; otherwise, let $\ell \geq 1$ be such that $L = 128(\ell - 1) + r$ where $1 \leq r \leq 128$. Write the message as a concatenation of ℓ strings, i.e., $M = M_0 || \cdots || M_{\ell-1}$ such that $M_0, \ldots, M_{\ell-2}$ each have length 128 bits and $M_{\ell-1}$ has length r bits. For $i = 0, \ldots, \ell - 2$, define $C_i = M_i || 1$, and $C_{\ell-1} = M_{\ell-1} || 10^{128-r}$. This ensures that the length of C_i is 129 bits for $i = 0, \ldots, \ell - 1$. Let $\mathsf{format}(M)$ be the map from a message M to $(C_0, \ldots, C_{\ell-1})$.

From the above description, C_i is the binary representation (written with the least significant bit on the left) of an integer which is less than 2^{129}. For convenience of notation, we will identify the

binary string C_i with the integer it represents. Note that the C_i's cannot take all the values in the set $\{0, \ldots, 2^{129} - 1\}$; in particular, none of the C_i's can be zero.

The Poly1305 hash function uses a key R which is an element of $\mathbb{F}_p$. The specification of Poly1305 requires some of the bits R to be set to zero. This was done for efficiency purposes. For the SIMD implementation that we consider, the setting of certain bits of R to be zero does not either help or hamper the efficiency. So, we skip the details of the exact form of R which are given in [3].

The Poly1305 hash function is defined as follows. Given a message M consisting of $L \geq 0$ bits and a key $R \in \mathbb{F}_p$, the output is defined to be the following.

$$\mathsf{Poly1305}_R(M) \quad = \quad C_0 R^{\ell} + C_1 R^{\ell-1} + \cdots + C_{\ell-2} R^2 + C_{\ell-1} R \tag{1}$$

where $(C_0, \ldots, C_{\ell-1}) = \mathsf{format}(M)$. Since the map $\mathsf{format} : M \to (C_0, \ldots, C_{\ell-1})$ is injective, by an abuse of notation, instead of writing $\mathsf{Poly1305}_R(M)$, we will write $\mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1})$. Note that if M is the empty string, i.e., $L = 0$, then $\ell = 0$ and so $\mathsf{Poly1305}_R(M)$ is 0 (the zero element of $\mathbb{F}_p$).

3 Goll-Gueron SIMD Implementation

Given $C_0, \ldots, C_{\ell-1} \in \mathbb{F}_p$ and R, we define $\mathsf{poly}_R(C_0, \ldots, C_{\ell-1})$ as follows.

$$\mathsf{poly}_R(C_0, \ldots, C_{\ell-1}) \quad = \quad C_0 R^{\ell-1} + \cdots + C_{\ell-2} R + C_{\ell-1}. \tag{2}$$

So, $\mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1}) = R \cdot \mathsf{poly}_R(C_0, \ldots, C_{\ell-1})$.

The definition of poly in (2) permits the computation of the output using Horner's rule in the following manner.

$$\mathsf{poly}_R(C_0, \ldots, C_{\ell-1}) \quad = \quad ((\cdots(((C_0 R + C_1)R) + C_2)R + \cdots)R + C_{\ell-1}. \tag{3}$$

This requires $\ell - 1$ multiplications and $\ell - 1$ additions over $\mathbb{F}_p$. As a result, Poly1305 can be computed using ℓ multiplications and $\ell - 1$ additions over $\mathbb{F}_p$.

Horner's rule is a sequential method of evaluation. One way to exploit parallelism in the computation is to divide the sequence $(C_0, \ldots, C_{\ell-1})$ into $d \geq 2$ subsequences and apply Horner's rule to each of the subsequence. This allows alternatively performing d simultaneous multiplications and d simultaneous additions. Such a strategy has been called d-decimated Horner evaluation [5]. Goll and Gueron [6] described SIMD implementations of Poly1305 based on d-decimated Horner evaluation. They considered two values of d, namely, $d = 4$ and $d = 8$ leading to 4-way and 8-way SIMD implementations respectively. We provide details for $d = 4$, the case of $d = 8$ being similar.

Let $\rho = \ell \bmod 4$ and $\ell' = (\ell - \rho)/4$. The computation of $\mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1})$ can be done in the following manner. Let

$$\begin{aligned} P \quad = \quad & R^4 \ \mathsf{poly}_{R^4}(C_0, C_4, C_8, \ldots, C_{4\ell'-4}) \\ & + R^3 \ \mathsf{poly}_{R^4}(C_1, C_5, C_9, \ldots, C_{4\ell'-3}) \\ & + R^2 \ \mathsf{poly}_{R^4}(C_2, C_6, C_{10}, \ldots, C_{4\ell'-2}) \\ & + R \ \mathsf{poly}_{R^4}(C_3, C_7, C_{11}, \ldots, C_{4\ell'-1}) \end{aligned} \tag{4}$$

Then

$$\mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1}) \quad = \quad \begin{cases} P & \text{if } \rho = 0; \\ R \ \mathsf{poly}_R(P + C_{\ell-\rho}, C_{\ell-\rho+1}, \ldots, C_{\ell-1}) & \text{if } \rho > 0. \end{cases} \tag{5}$$

Define

$$\begin{aligned} \mathbf{R} &= \left(R^4, R^3, R^2, R\right)^T; \\ \mathbf{R}_4 &= (R^4, R^4, R^4, R^4)^T; \\ \mathbf{C}_{4i} &= (C_{4i}, C_{4i+1}, C_{4i+2}, C_{4i+3})^T; \text{ for } i = 0, 1, \ldots, \ell' - 1. \end{aligned} \tag{6}$$

Here the subscript $()^T$ denotes the transpose of the corresponding vector.

The computation in (5) is described in vector form in Algorithm 1. In the description of Algorithm 1, a temporary vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ is used and $\circ$ denotes the Hadamard (i.e., component-wise) product of vectors. The quantity P is a temporary field element.

Algorithm 1: Structure of Goll-Gueron 4-way vectorisation of Poly1305 computation. Refer to (6) for the definition of the vector quantities.

Input: $(C_0, \ldots, C_{\ell-1})$
Output: $\mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1})$

$\mathbf{T} \leftarrow \mathbf{C}_0$;
for $i = 1$ *to* $\ell' - 1$ **do**
  $\mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{C}_{4i}$
$\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
$P = T_0 + T_1 + T_2 + T_3$
if $\rho > 0$ **then**
  $P \leftarrow R\ \mathsf{poly}_R(P + C_{\ell-\rho}, C_{\ell-\rho+1}, \ldots, C_{\ell-1})$
return P

3.1 Vector Multiplication

Recall that $p = 2^{130} - 5$ and so any element of $\mathbb{F}_p$ can be represented using 130 bits. Let $\theta = 2^{26}$. An element $X \in \mathbb{F}_p$ can be written in base θ as follows.

$$X = x_0 + x_1\theta + x_2\theta^2 + x_3\theta^3 + x_4\theta^4$$

where $0 \leq x_0, \ldots, x_4 \leq 2^{26} - 1$. Then $(x_4, x_3, x_2, x_1, x_0)$ is called a 5-limb representation of X.

Given 5-limb representations $(x_4, x_3, x_2, x_1, x_0)$ and $(y_4, y_3, y_2, y_1, y_0)$ of X and Y respectively, the product $X \cdot Y \bmod p$ is computed in two steps.

Multiplication step: $Z = z_0 + z_1\theta + \cdots + z_4\theta^4$ is obtained where $z_0, \ldots, z_4$ are defined as follows.

$$\begin{aligned} z_0 &= x_0 \cdot y_0 + 5 \cdot x_1 \cdot y_4 + 5 \cdot x_2 \cdot y_3 + 5 \cdot x_3 \cdot y_2 + 5 \cdot x_4 \cdot y_1 \\ z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + 5 \cdot x_2 \cdot y_4 + 5 \cdot x_3 \cdot y_3 + 5 \cdot x_4 \cdot y_2 \\ z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + 5 \cdot x_3 \cdot y_4 + 5 \cdot x_4 \cdot y_3 \\ z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 + 5 \cdot x_4 \cdot y_4 \\ z_4 &= x_0 \cdot y_4 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 + x_4 \cdot y_0. \end{aligned} \tag{7}$$

Note that each z_i is less than 2^{64}. By $\mathsf{mult}((x_4, \ldots, x_0), (y_4, \ldots, y_0))$ we will denote the vector $(z_0, \ldots, z_4)$.

Reduction step: $W = w_0 + w_1\theta + \cdots + w_4\theta^4$ is obtained such that $W \equiv Z \bmod 4$ and each w_i can be represented using either 26 or 27 bits. By $\mathsf{reduce}(z_4, \ldots, z_0)$ we will denote the vector $(w_4, \ldots, w_0)$. For the details of the reduction step, we refer to [3].

Suppose X is a fixed quantity and the product $X \cdot Y \bmod p$ is required to be computed. Note that the computation in (7) is helped by pre-computing and storing $(5 \cdot x_4, 5 \cdot x_3, 5 \cdot x_2, 5 \cdot x_1)$ along with the 5-limb representation $(x_4, x_3, x_2, x_1, x_0)$ of X.

Vector multiplication: Algorithm 1 requires the vector multiplication

$$\mathbf{R}_4 \circ \mathbf{T} = (R^4 \cdot T_0, R^4 \cdot T_1, R^4 \cdot T_2, R^4 \cdot T_3)^T.$$

Note that the multiplication in Step 3 of Algorithm 1 has one of the operands to be fixed to $\mathbf{R}_4$ while the other operand changes. Goll and Gueron [6] presented a very efficient SIMD algorithm to do this multiplication.

The vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ has four elements of $\mathbb{F}_p$. Each of these elements has a 5-limb representation. Let $(t_{i,4}, \ldots, t_{i,0})$ be the 5-limb representation of T_i, $i = 0, 1, 2, 3$. So, a total of 20 26-bit quantities are required to store $\mathbf{T}$. Since intermediate results are not fully reduced, some of the $t_{i,j}$'s can be 27-bit quantities. Let $(r_4, \ldots, r_0)$ be the 5-limb representation of R^4. Also, the vector $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ is stored.

The 4-way SIMD implementation of Goll and Gueron [6] uses 256-bit words. Each 256-bit word is considered to be 8 32-bit words. So, the 20 26-bit quantities of $\mathbf{T}$ can be stored in 3 256-bit words. The vectors $(r_4, \ldots, r_0)$ and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ can be stored in 2 256-bit words. The multiplication $\mathbf{W} = \mathbf{R}_4 \circ \mathbf{T}$ consists of two steps.

Vector multiplication step: This step takes as input the 3 256-bit words representing $\mathbf{T} = (T_0, T_1, T_2, T_3)$ and the 2 256-bit words representing $(r_4, \ldots, r_0)$ and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$. It produces as output 5 256-bit words $S_0, \ldots, S_4$, where $S_i = (s_{i,3}, s_{i,2}, s_{i,1}, s_{i,0})$ and each $s_{i,j}$ is a 64-bit word. Further, $(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$ is $\mathsf{mult}((r_4, \ldots, r_0), (t_{j,4}, \ldots, t_{j,0}))$ for $j = 0, \ldots, 3$. Let $\mathbf{S} = (S_0, \ldots, S_4)$. By $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ we will denote $\mathbf{S}$.

Vector reduction step: This step takes as input $S_0, \ldots, S_4$ and produces as output 3 256-bit words which stores the 20 26-bit (or 27-bit) words of the result $\mathbf{W} = (W_0, W_1, W_2, W_3)$. Let $W_j = (w_{j,4}, \ldots, w_{j,0})$, $j = 0, 1, 2, 3$. Then $(w_{j,4}, \ldots, w_{j,0})$ is the output of $\mathsf{reduce}(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$. By $\mathsf{vecReduce}(S)$ we will denote $\mathbf{W}$.

In terms of the above notation, the computation $\mathbf{W} = \mathbf{R}^4 \circ \mathbf{T}$ consists of the following two steps: $\mathbf{S} \leftarrow \mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$; $\mathbf{W} \leftarrow \mathsf{vecReduce}(\mathbf{S})$. Note that $\mathbf{T}$ is stored in 3 256-bit words and the output $\mathbf{W}$ is also stored in 3 256-bit words. This ensures that the same multiplication algorithm can be applied to multiply $\mathbf{R}_4$ and $\mathbf{W}$, and so on.

We provide the top level schematics of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ and $\mathsf{vecReduce}(\mathbf{S})$. The 5-limb representation $(r_4, r_3, r_2, r_1, r_0)$ of R^4 and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ are represented in 2 256-bit words in the following manner.

$\mathtt{r_0}$	$5 \cdot r_4$	$5 \cdot r_3$	$5 \cdot r_2$	r_4	r_3	r_2	r_1	r_0
$\mathtt{r_1}$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$

The vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ is stored in 3 256-bit words in the following manner, where `x` denotes an undetermined quantity that is not used in the algorithm.

$\mathtt{t_0}$	$t_{3,2}$	$t_{3,0}$	$t_{2,2}$	$t_{2,0}$	$t_{1,2}$	$t_{1,0}$	$t_{0,2}$	$t_{0,0}$
$\mathtt{t_1}$	$t_{3,3}$	$t_{3,1}$	$t_{2,3}$	$t_{2,1}$	$t_{1,3}$	$t_{1,1}$	$t_{0,3}$	$t_{0,1}$
$\mathtt{t_2}$	x	$t_{3,4}$	x	$t_{2,4}$	x	$t_{1,4}$	x	$t_{0,4}$

Name of Intrinsic	Count
_mm256_mul_epu32	25
_mm256_set_epi32	1
_mm256_add_epi64	20
_mm256_permutevar8x32_epi32	9
_mm256_permute4x64_epi64	4

Table 1: Operation counts for $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ used in [6].

The Intel AVX2 implementation of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$, uses a number of SIMD permutation operations on $\mathtt{t_0}, \mathtt{t_1}$ and $\mathtt{t_2}$ followed by 32-bit SIMD multiplication operations with $\mathtt{r_0}$ and $\mathtt{r_1}$ and 64-bit SIMD operations to accumulate the results. Finally, the result of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ is $(S_0, \ldots, S_4)$ and is stored as follows.

S_0	$s_{0,3}$	$s_{0,2}$	$s_{0,1}$	$s_{0,0}$
S_1	$s_{1,3}$	$s_{1,2}$	$s_{1,1}$	$s_{1,0}$
S_2	$s_{2,3}$	$s_{2,2}$	$s_{2,1}$	$s_{2,0}$
S_3	$s_{3,3}$	$s_{3,2}$	$s_{3,1}$	$s_{3,0}$
S_4	$s_{4,3}$	$s_{4,2}$	$s_{4,1}$	$s_{4,0}$

The number of different operations (in Intel intrinsics) required by $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ is given in Table 1.

The $\mathsf{vecReduce}(\mathbf{S})$ implementation takes as input the five 256-bit words $S_0, \ldots, S_4$ and produces as output the vector $\mathbf{W} = (W_0, W_1, W_2, W_3)$ stored in 3 256-bit words $\mathtt{w_0}, \mathtt{w_1}$ and $\mathtt{w_2}$ as follows.

$\mathtt{w_0}$	$w_{3,2}$	$w_{3,0}$	$w_{2,2}$	$w_{2,0}$	$w_{1,2}$	$w_{1,0}$	$w_{0,2}$	$w_{0,0}$
$\mathtt{w_1}$	$w_{3,3}$	$w_{3,1}$	$w_{2,3}$	$w_{2,1}$	$w_{1,3}$	$w_{1,1}$	$w_{0,3}$	$w_{0,1}$
$\mathtt{w_2}$	x	$w_{3,4}$	x	$w_{2,4}$	x	$w_{1,4}$	x	$w_{0,4}$

The evaluation in Step 7 of Algorithm 1 is also done using vector operations. This is not clearly described in the paper [6] and can be understood from the accompanying code. Since Step 7 is not relevant to our algorithm we do not describe the details of its computation.

4 A New SIMD Implementation of Poly1305

Algorithm 1 implements the computation in (4). If $4|\ell$ (i.e., $\rho = 0$), then the 4-way SIMD computation proceeds uniformly throughout. However, if $4 \not| \ell$, then the 4-way SIMD computation in Algorithm 1 proceeds uniformly for ℓ' steps. Additionally, the computation in Step 7 is required making the computation non-uniform. For short messages, this leads to a significant penalty.

By making a simple modification, it can be ensured that the 4-way SIMD proceeds uniformly throughout. As before, let $\rho = \ell \bmod 4$. If $\rho = 0$, let $m = \ell$; and if $\rho > 0$, let $m = \ell + 4 - \rho$. Given the sequence $(C_0, \ldots, C_{\ell-1})$ obtained as $\mathsf{format}(M)$, define the sequence $(D_0, \ldots, D_{m-1})$ where if $\rho = 0$, then $D_i = C_i$ for $i = 0, \ldots, \ell - 1$ and if $\rho > 0$, then

$$\begin{array}{rcll} D_i & = & 0 & \text{for } i = 0, \ldots, 3 - \rho; \\ D_i & = & C_{i-4+\rho} & \text{for } i = 4 - \rho, \ldots, m - 1. \end{array} \tag{8}$$

In the definition $D_i = 0$, the '0' is the zero element of $\mathbb{F}_p$ and not the bit 0. The zero element of $\mathbb{F}_p$ is represented in binary using a zero block which is a binary string consisting of 129 zero bits.

In other words, the sequence $(C_0, \ldots, C_{\ell-1})$ is prepended using a minimum number of zero blocks to make the length a multiple of 4. Since the initial zeros have no effect on the computation of $\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1})$, we have

$$\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1}) \;=\; \mathsf{Poly1305}_R(C_0, \ldots, C_{\ell-1}). \tag{9}$$

Let $m' = m/4$. The computation of $\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1})$ can be written as follows.

$$\begin{aligned} \mathsf{Poly1305}_R(D_0, \ldots, D_{m-1}) \;=\;& R^4\ \mathsf{poly}_{R^4}(D_0, D_4, \ldots, D_{m-4}) \\ &+ R^3\ \mathsf{poly}_{R^4}(D_1, D_5, \ldots, D_{m-3}) \\ &+ R^2\ \mathsf{poly}_{R^4}(D_2, D_6, \ldots, D_{m-2}) \\ &+ R^1\ \mathsf{poly}_{R^4}(D_3, D_7, \ldots, D_{m-1}) \end{aligned} \tag{10}$$

In a manner similar to (6), define

$$\begin{aligned} \mathbf{R} &= \left(R^4, R^3, R^2, R\right)^T; \\ \mathbf{R}_4 &= (R^4, R^4, R^4, R^4)^T; \\ \mathbf{D}_{4i} &= (D_{4i}, D_{4i+1}, D_{4i+2}, D_{4i+3})^T;\ \text{for } i = 0, \ldots, m'-1. \end{aligned} \tag{11}$$

The computation in (10) is described in vector form in Algorithm 2.

Algorithm 2: Structure of the new 4-way vectorisation of Poly1305 computation. Refer to (11) for the definition of the vector quantities.

Input: $(D_0, \ldots, D_{\ell-1})$
Output: $\mathsf{Poly1305}_R(D_0, \ldots, D_{\ell-1})$

1 $\mathbf{T} \leftarrow \mathbf{D}_0$;
2 **for** $i = 1$ *to* $m' - 1$ **do**
3 $\quad \mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{D}_{4i}$
4 $\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
5 return $T_0 + T_1 + T_2 + T_3$

In Algorithm 2, the entire computation can be performed using 4-way SIMD operations. In other words, by prepending 0's, the structure of the computation becomes balanced. It is possible to execute all the multiplications arising in Step 3 using $\mathsf{vecMult}(\cdot, \cdot)$ followed by $\mathsf{vecReduce}(\cdot)$.

For the case $\rho = 0$, the Step 7 of Algorithm 1 is not executed. In this case, Algorithms 1 and 2 become the same. For $\rho = 3$, there is a performance improvement of Algorithm 2 over Algorithm 1. For the cases of $\rho = 1$ and $\rho = 2$, the situation is more subtle. Directly using vecMult for the first multiplication in Algorithm 2 does not necessarily lead to speed gains. We address this issue in the next section.

Remark: We have described Algorithm 2 for 4-decimated Horner computation. The same idea easily extends to d-decimated Horner computation for any $d \geq 2$.

4.1 Improved Initial Multiplication

In Algorithm 2, the first multiplication is $\mathbf{R}_4 \circ \mathbf{D}_0$. Consider $\mathbf{D}_0 = (D_0, D_1, D_2, D_3)^T$. Depending on the value of ρ, there are four cases.

$$\mathbf{D}_0 \quad = \quad \begin{cases} (C_0, C_1, C_2, C_3)^T & \text{if } \rho = 0; \\ (0, 0, 0, C_0)^T & \text{if } \rho = 1; \\ (0, 0, C_0, C_1)^T & \text{if } \rho = 2; \\ (0, C_0, C_1, C_2)^T & \text{if } \rho = 3. \end{cases} \tag{12}$$

Suppose $\rho = 1$ so that $\mathbf{D}_0 = (0, 0, 0, C_0)$. Let the 5-limb representation of C_0 be given by $(c_{0,4}, \ldots, c_{0,0})$. Consider the schematic of the operation vecMult as discussed in Section 3.1. The representation of $\mathbf{D}_0$ in the 3 256-bit words $\mathtt{t_0}, \mathtt{t_1}$ and $\mathtt{t_2}$ will look as follows (where x is a don't care value).

$\mathtt{t_0}$	$c_{0,2}$	$c_{0,0}$	0	0	0	0	0	0
$\mathtt{t_1}$	$c_{0,3}$	$c_{0,1}$	0	0	0	0	0	0
$\mathtt{t_2}$	x	$c_{0,4}$	x	0	x	0	x	0

Since a lot of entries in the above representation are zeros, applying the Goll-Gueron vecMult operation to $\mathbf{R}_4$ and this $\mathbf{D}_0$ will result in the execution of a number of 32-bit multiplication operations whose results are known to be zero. By using a different representation for $\mathbf{D}_0$ and a different multiplication algorithm, it is possible to obtain the desired output using a substantially lower number of 32-bit SIMD multiplication operations. This leads to speed improvement.

A similar analysis shows that it is possible to obtain speed improvement also for the case $\rho = 2$ for which $\mathbf{D}_0 = (0, 0, C_0, C_1)$. When $\rho = 3$, $\mathbf{D}_0 = (0, C_0, C_1, C_2, C_3)$, and in this case, the number of zeros is not sufficient to provide any improvement by using a multiplication algorithm different from vecMult. Below we provide the top level schematics of the improved initial multiplication algorithms for the cases of $\rho = 1$ and $\rho = 2$.

Representation of $\mathbf{D}_0$ for the case $\rho = 1$: In this case, $\mathbf{D}_0 = (0, 0, 0, C_0)$ is represented using 2 256-bit words as follows.

$\mathtt{t_0}$	0	$c_{0,3}$	0	$c_{0,2}$	0	$c_{0,1}$	0	$c_{0,0}$
$\mathtt{t_1}$	0	$c_{0,4}$	0	0	0	0	0	0

The 5 256-bit words holding the output of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{D}_0)$ in this case will have the following form.

S_0	$s_{0,3}$	0	0	0
S_1	$s_{1,3}$	0	0	0
S_2	$s_{2,3}$	0	0	0
S_3	$s_{3,3}$	0	0	0
S_4	$s_{4,3}$	0	0	0

Representation of $\mathbf{D}_0$ for the case $\rho = 2$: In this case, $\mathbf{D}_0 = (0, 0, C_0, C_1)$ is represented using 2 256-bit words as follows.

$\mathtt{t_0}$	$c_{0,3}$	$c_{1,3}$	$c_{0,2}$	$c_{1,2}$	$c_{0,1}$	$c_{1,1}$	$c_{0,0}$	$c_{1,0}$
$\mathtt{t_1}$	0	$c_{1,4}$	0	$c_{0,4}$	0	0	0	0

The 5 256-bit words holding the output of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{D}_0)$ in this case will have the following form.

Name of Intrinsic	Count for $\ell \bmod 4 = 1$	Count for $\ell \bmod 4 = 2$
_mm256_mul_epu32	7	13
_mm256_set_epi32	1	6
_mm256_set_epi64x	7	2
_mm256_add_epi64	7	11
_mm256_permutevar8x32_epi32	7	10
_mm256_permute4x64_epi64	9	7
_mm256_unpacklo_epi64	2	3
_mm256_unpackhi_epi64	2	3
_mm256_blend_epi32	2	1
_mm256_shuffle_epi32	-	6

Table 2: Operation counts for the modified multiplication algorithms for the cases $\ell \bmod 4 = 1$ or 2.

S_0	$s_{0,3}$	$s_{0,2}$	0	0
S_1	$s_{1,3}$	$s_{1,2}$	0	0
S_2	$s_{2,3}$	$s_{2,2}$	0	0
S_3	$s_{3,3}$	$s_{3,2}$	0	0
S_4	$s_{4,3}$	$s_{4,2}$	0	0

In both the cases, the representation of $\mathbf{R}_4$ using 2 256-bit words $\mathtt{r_0}$ and $\mathtt{r_1}$ remain the same as in Section 3.1. The multiplication algorithms for the above two cases apply SIMD permutations, 32-bit SIMD multiplications and 64-bit SIMD additions to produce the required output $\mathbf{S}$ in 5 256-bit words as shown above. The operation counts for the two cases of $\rho = 1$ and $\rho = 2$ are shown in Table 2. Comparing Tables 1 and 2, we see that the number of 32-bit SIMD multiplications and 64-bit SIMD additions come down substantially while the counts of the other operations are similar. It is due to the decrease in the number of operations that a speed-up is obtained by using the modified initial multiplication algorithm.

The vecReduce algorithm mentioned in Section 3.1 is applied to $\mathbf{S}$ to obtain the result $\mathbf{R}_4 \circ \mathbf{D}_0$. The output of vecReduce is in the form of 3 256-bit words which is stored in $\mathbf{T}$. The further multiplications $\mathbf{R}_4 \circ \mathbf{T}$ in the loop at Step 1 of Algorithm 2 are performed using the algorithm vecMult and vecReduce.

Remark: For the case $\rho = 2$, there are several variants of the initial multiplication algorithm which avoid multiplications by zero. For all such variants the number of _mm256_mul_epu32 is 13. In Table 2, we report the operation counts for the variant having the least number of operations. For practical implementation, however, we have found another variant to provide somewhat better performance.

5 Implementation and Comparison

We have implemented the SIMD strategy given in Algorithm 2 for evaluation of the Poly1305 hash function. This implementation consisted of modifying the Intel intrinsics code[3] implementing the SIMD strategy in [6]. Portions of the code were used without any change. In particular, the basic 4-way multiplication routine of [6] has been directly used. On the other hand, the improved initial multiplication algorithms are new to our SIMD strategy and had to be implemented.

[3]The code package was provided to us by Gueron.

Performance has been measured in terms of number of machine cycles per byte under the same conditions as mentioned in [6]: Intel Turbo Boost Technology, Intel Hyper-Threading Technology and Intel Speedstep Technology were disabled. The measurement strategy of cache warming and number of iterations that were used is the same as that of [6].

Measurements were made on the following platforms.

- Haswell: Intel Core i7-4790 CPU @ 3.60GHz x 8; running Ubuntu 18.04.2 LTS (64-bit); gcc version 7.4.0.
- Skylake: Intel Core i7-6500U CPU @ 2.50GHz x 2; running Ubuntu 14.04 LTS (64-bit); gcc version 5.5.0.
- Kaby Lake: Intel Core i7-7700U CPU @ 3.60GHz x 4; running Ubuntu 18.04 LTS (64-bit); gcc version 7.3.0.

In all cases, measurements were made on a single core of the specified machines; compiler flags '-mavx2', '-O3' and '-fomit-frame-pointer' were used during compilation.

Message length: For measuring performance and comparison to [6], we considered messages with lengths up to 4KB[4]. If the number of 16-byte blocks in the padded message is a multiple of 4, then the new code becomes exactly the Goll-Gueron code. Consequently, there is no difference of performance for such message lengths.

In view of the above, for the purpose of comparing the performance of the new code to the Goll-Gueron code, we considered message lengths from 49 bytes to 4000 bytes which are not divisible by 64. For each message length, we have taken measurements of both the Goll-Gueron code and the new code. A top-level summary of the comparison is as follows.

- Haswell: speed-up has been obtained for message lengths cases.
- Skylake: speed-up has been obtained in 94.97% cases; in 4.08% cases, the performances of both the codes were the same; in 0.95% cases, the new code has shown a slight slowdown.
- Kaby Lake: speed-up has been obtained in 96.38% cases; in 3.29% cases, the performances of both the codes were the same; in 0.33% cases, the new code has shown a slight slowdown.

Table 3 provides a summary of the maximum and average speed-ups for the three platforms for three different ranges of message lengths. Detailed plots of performance improvements are provided in Appendix A.

6 Conclusion

In this work, we have proposed a simple modification to the previous Goll-Gueron strategy for SIMD implementation of the Poly1305 algorithm. Implementation of the modified algorithm shows noticeable speed improvements on modern Intel processors for short messages when the number of blocks is not a multiple of 4. While we have tried to perform code optimisation to improve the speed, there is the possibility that the code can be optimised further leading to improved speed-ups.

[4]See `http://www.caida.org/data/passive/trace_stats/nyc-A/2019/equinix-nyc.dirA.20190117-130000.UTC.df.xml` of the Center for Applied Internet Data Analysis for the relevance of short messages in IPv4 and IPv6 traffic. Accessed on 27th June, 2019.

Table 3: A summary of the comparative analysis of the new code with the code of [6].

Processor	Range	Maximum Speed-up	Average Speed-up
Haswell	49B - 1000B	31.64	13.68
	1001B - 2000B	18.60	10.43
	2001B - 4000B	13.00	6.39
Skylake	49B - 1000B	33.04	13.84
	1001B - 2000B	14.70	6.42
	2001B - 4000B	10.71	3.53
Kaby Lake	49B - 1000B	34.35	15.31
	1001B - 2000B	16.03	7.50
	2001B - 4000B	19.53	3.89

Acknowledgement

We would like to thank Shay Gueron for kindly sharing the code associated with [6] with us.

References

[1] Daniel J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, January 2008, `http://cr.yp.to/chacha/chacha-20080128.pdf`.

[2] Daniel J. Bernstein. The Salsa20 family of stream ciphers. `http://cr.yp.to/papers.html#salsafamily`. Document ID: 31364286077dcdff8e4509f9ff3139ad. Date: 2007.12.25.

[3] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[4] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012.

[5] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. *IACR Trans. Symmetric Cryptol.*, 2017(1):106–128, 2017.

[6] Martin Goll and Shay Gueron. Vectorization of Poly1305 message authentication code. In *2015 12th International Conference on Information Technology - New Generations*, pages 145–150. IEEE, April 2015. 10.1109/ITNG.2015.28.

A Details of Performance Comparison

For Haswell, Figure 1 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 2 to 4. For Skylake, Figure 5 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 6 to 8. For Kaby Lake, Figure 9 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 10 to 12.

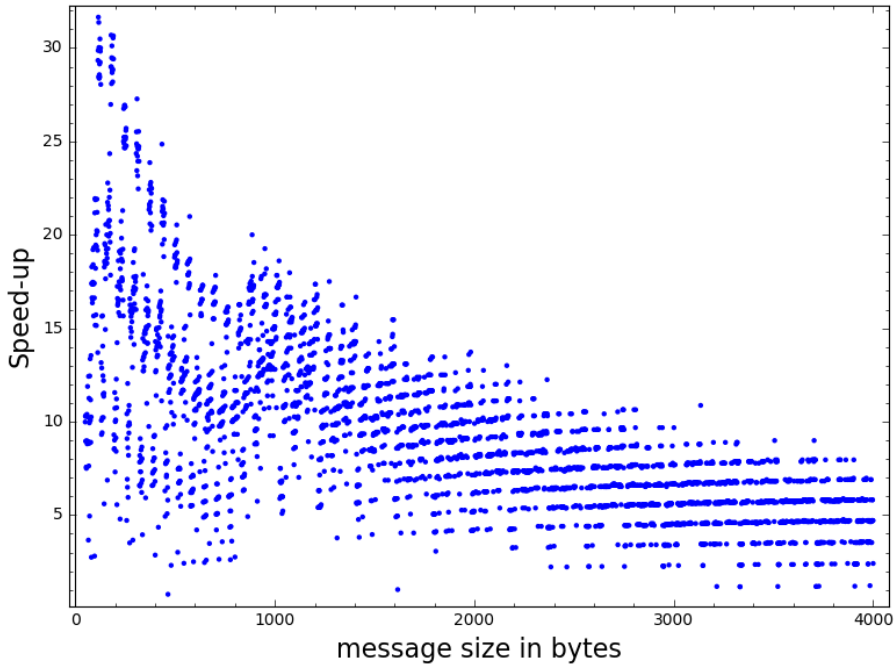

Figure 1: Speed-up vs message size in bytes for Haswell.

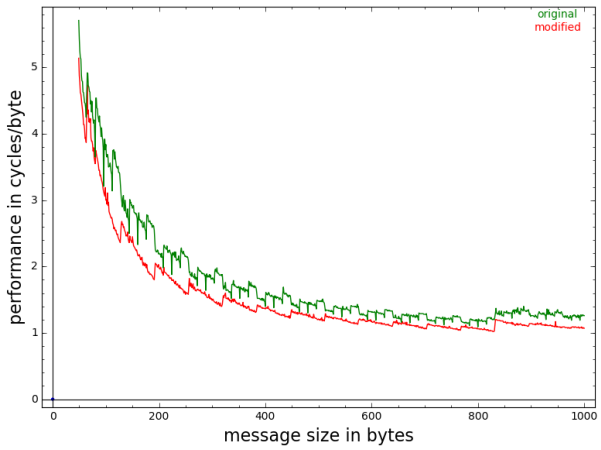

Figure 2: cycles/byte vs message size in bytes (49 - 1000 bytes) for Haswell.

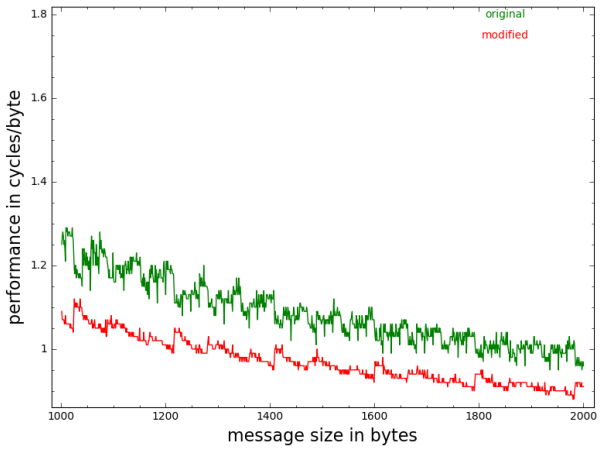

Figure 3: cycles/byte vs message size in bytes (1001 - 2000 bytes) for Haswell.

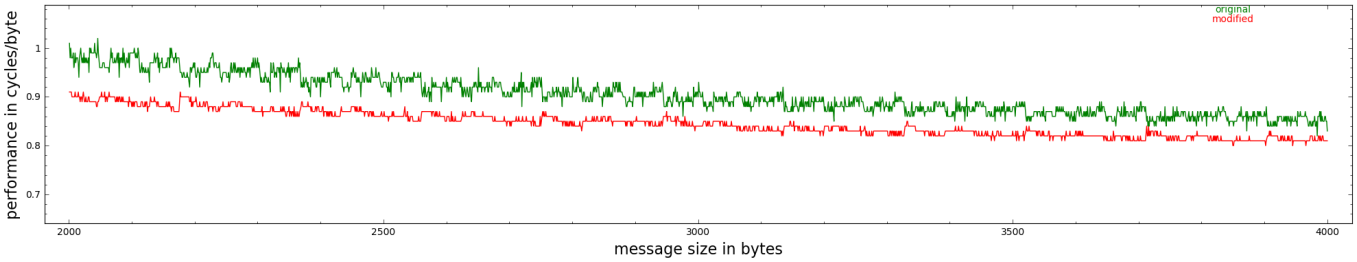

Figure 4: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Haswell.

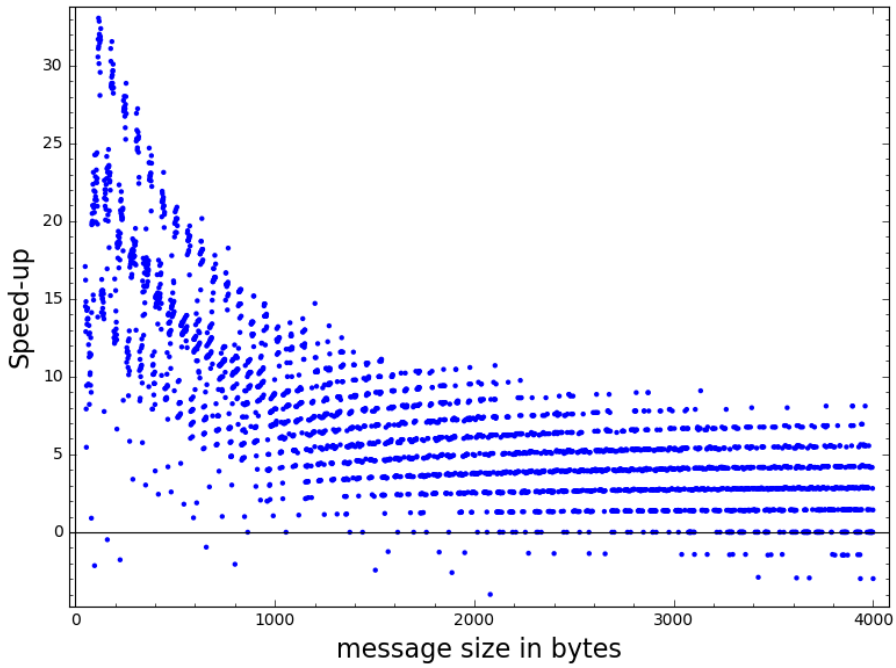

Figure 5: Speed-up vs message size in bytes graph for Skylake.

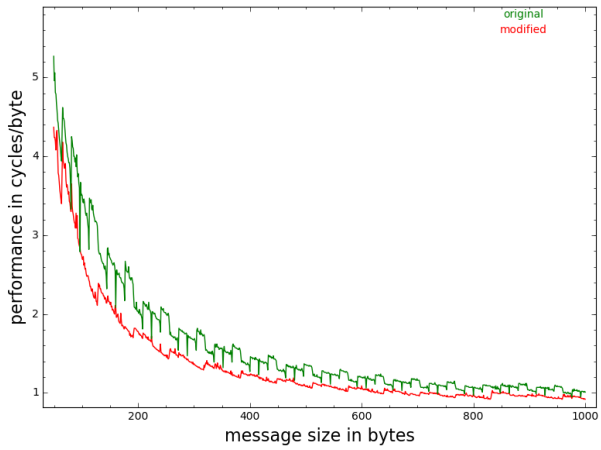

Figure 6: cycles/byte vs message size in bytes (49 - 1000 bytes) graph for Skylake.

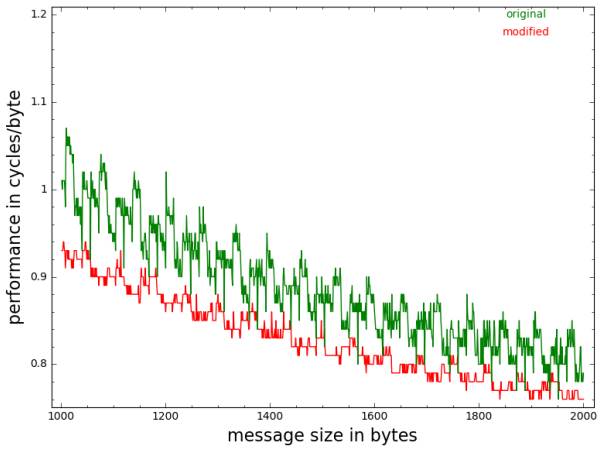

Figure 7: cycles/byte vs message size in bytes (1001 - 2000 bytes) graph for Skylake.

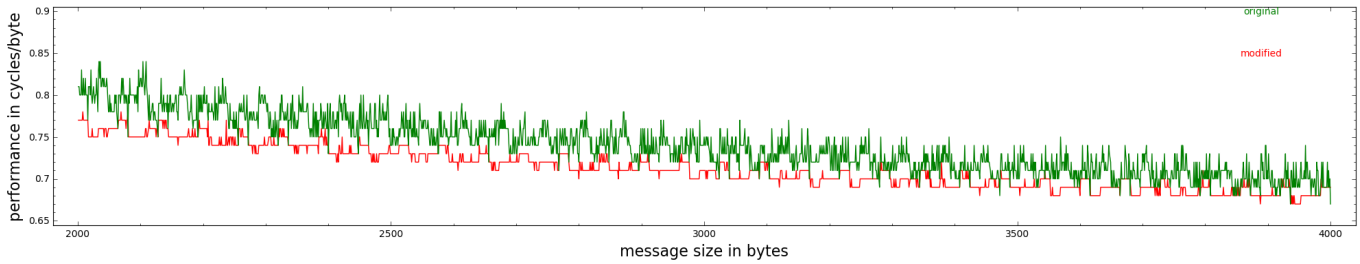

Figure 8: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Skylake.

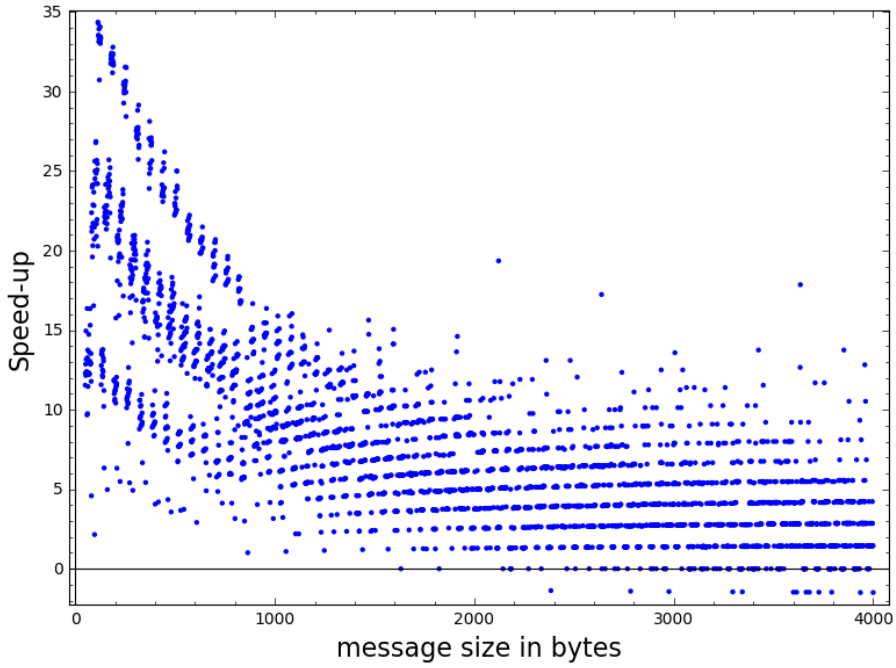

Figure 9: Speed-up vs message size in bytes graph for Kaby Lake.

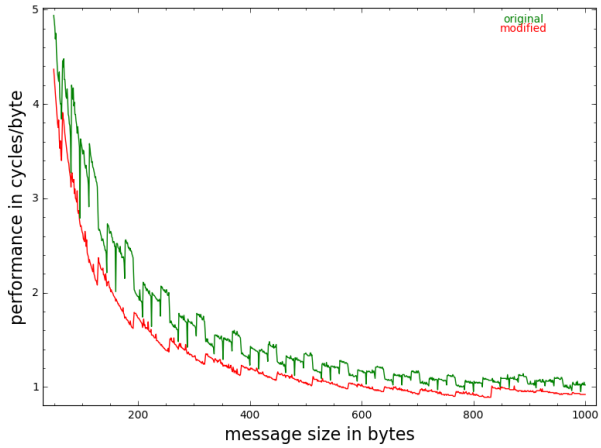

Figure 10: cycles/byte vs message size in bytes (49 - 1000 bytes) graph for Kaby Lake.

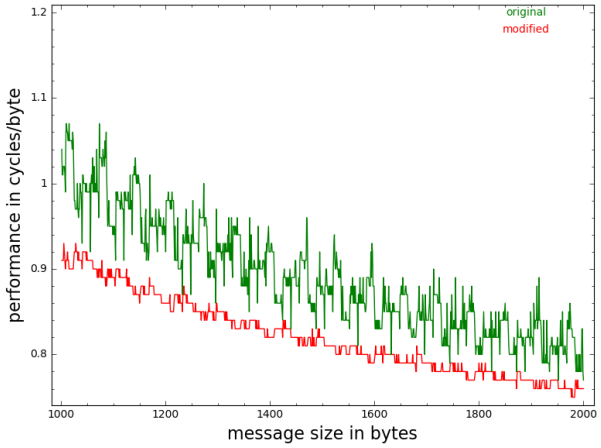

Figure 11: cycles/byte vs message size in bytes (1001 - 2000 bytes) graph for Kaby Lake.

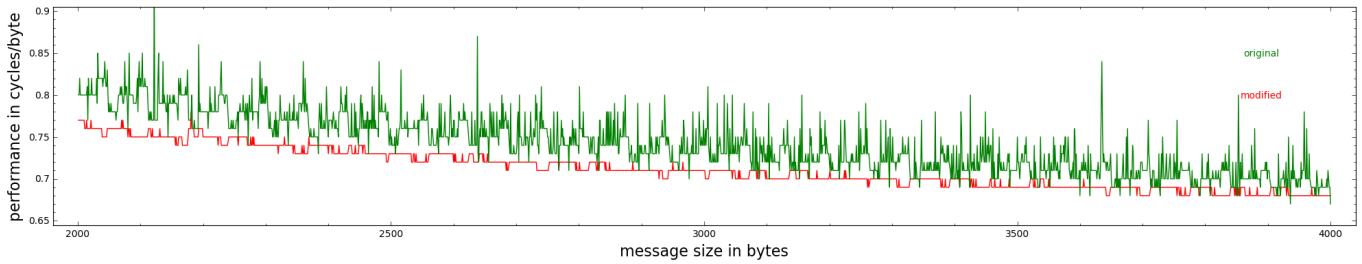

Figure 12: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Kaby Lake.