9-1-2020

Improved SIMD implementation of Poly1305

Sreyosi Bhattacharyya
Indian Statistical Institute, Kolkata

Palash Sarkar
Indian Statistical Institute, Kolkata

IET Information Security

Research Article

Improved SIMD implementation of Poly1305

ISSN 1751-8709
Received on 20th November 2019
Revised 6th March 2020
Accepted on 24th March 2020
E-First on 27th April 2020
doi: 10.1049/iet-ifs.2019.0605
www.ietdl.org

Sreyosi Bhattacharyya[1], *Palash Sarkar*[1] ✉

[1]*Applied Statistics Unit, Indian Statistical Institute, 203, B.T. Road, Kolkata 700108, India*

✉ *E-mail: palash@isical.ac.in*

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 was proposed by Goll and Gueron in 2015. The authors 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.

1 Introduction

Confidentiality and integrity of data flowing through the Internet are of paramount importance. The Transport Layer Security (TLS) protocol is used to provide security of Internet communications. TLS includes a variety of algorithms for different cryptographic functionalities. Among the algorithms which are part of TLS is Poly1305 [1] (in combination with ChaCha [2]). Apart from TLS, Poly1305 is part of various other cryptographic libraries: it has been standardised by the IETF and is part of the NaCl [3] library (in combination with Salsa20 [4]). Further, Google uses ChaCha-Poly1305 to secure communication between Chrome browsers on Android and Google websites. The Wikipedia page (https://en.wikipedia.org/wiki/Poly1305, accessed on 12 February 2020) for Poly1305 provides further details about the real-world deployment of Poly1305. Given the widespread use of Poly1305, efficient software implementation of the algorithm is an important practical issue.

Modern processors are moving towards providing vector instructions. These instructions allow single-instruction, multiple-data (SIMD) implementations of a variety of algorithms which often provide substantial speed gains over sequential implementations. The importance of vectorisation in modern processors has been highlighted in a post by Bernstein (https://groups.google.com/a/list.nist.gov/forum/ #!searchin/pqc-forum/vectorization%7Csort:date/pqc-forum/mmsH4k3j_1g/JfzP1EBuBQAJ, accessed on 12 February 2020).

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 [5]. They showed a way to divide the Poly1305 computation into d independent and parallel computation streams. Concrete implementations were provided in [5] 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 four blocks at a time. If the number of blocks in the message is a multiple of four, then the algorithm uniformly processes all the blocks. On the other hand, if the number of blocks is not a multiple of four, then at the end, the parallelism breaks down, and the tail of the message consisting of one to three blocks has 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 four. Then the processing of the blocks can be done four 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 three zero blocks would need to be prepended. Consider the initial 4-way multiplication. This consists of three zero blocks and one message block. 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 number of blocks are not multiples of four. Comparative speed measurements of the new algorithm and the Goll–Gueron algorithm have been made on the Haswell, Kaby Lake and Skylake processors.

Goll and Gueron [5] also describe an 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 [1] of `Poly1305`, the message is a sequence of bytes. The later work [5] 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 [5] coincides with the original description in [1]. Goll and Gueron [5] 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., $M = M_0 \parallel \cdots \parallel M_{\ell-1}$ such that $M_0, \ldots, M_{\ell-2}$ each has length 128 bits and $M_{\ell-1}$ has length r bits. For $i = 0, \ldots, \ell - 2$, define $C_i = M_i \parallel 1$, and $C_{\ell-1} = M_{\ell-1} \parallel 10^{128-r}$. This ensures that

```
    Input: $(C_0,\ldots,C_{\ell-1})$
    Output: $\mathsf{Poly1305}_R(C_0,\ldots,C_{\ell-1})$
1   $\mathbf{T} \leftarrow \mathbf{C}_0$;
2   for $i = 1$ to $\ell' - 1$ do
3       $\mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{C}_i$
4   $\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
5   $P = T_0 + T_1 + T_2 + T_3$
6   if $\rho > 0$ then
7       $P \leftarrow R\, \mathsf{poly}_R(P + C_{\ell-\rho}, C_{\ell-\rho+1},\ldots,C_{\ell-1})$
8   return $P$
```

Fig. 1 *Algorithm 1: Structure of Goll–Gueron 4-way vectorisation of* `Poly1305` *computation. Refer to (7) for the definition of the vector quantities*

the length of C_i is 129 bits for $i = 0,\ldots,\ell - 1$. Let `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 the 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 [1].

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 of $\mathtt{Poly1305}_R(M)$ is defined as follows:

$$\begin{aligned}&\mathtt{Poly1305}_R(M)\\&\quad = C_0R^{\ell} + C_1R^{\ell-1} + \cdots + C_{\ell-2}R^2 + C_{\ell-1}R\end{aligned} \tag{1}$$

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

$$\begin{aligned}&\mathtt{poly}_R(C_0,\ldots,C_{\ell-1})\\&\quad = C_0R^{\ell-1} + \cdots + C_{\ell-2}R + C_{\ell-1}.\end{aligned} \tag{2}$$

So

$$\begin{aligned}&\mathtt{Poly1305}_R(C_0,\ldots,C_{\ell-1})\\&\quad = R \cdot \mathtt{poly}_R(C_0,\ldots,C_{\ell-1}).\end{aligned} \tag{3}$$

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

$$\begin{aligned}&\mathtt{poly}_R(C_0,\ldots,C_{\ell-1})\\&\quad = ((\cdots(((C_0R + C_1)R) + C_2)R + \cdots)R + C_{\ell-1}.\end{aligned} \tag{4}$$

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 $d \geq 2$ subsequence 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 [6]. Goll and Gueron [5] 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 $\mathtt{Poly1305}_R(C_0,\ldots,C_{\ell-1})$ can be done in the following manner. Let

$$\begin{aligned}P = {}& R^4\, \mathtt{poly}_{R^4}(C_0, C_4, C_8, \ldots, C_{4\ell'-4})\\&+R^3\, \mathtt{poly}_{R^4}(C_1, C_5, C_9, \ldots, C_{4\ell'-3})\\&+R^2\, \mathtt{poly}_{R^4}(C_2, C_6, C_{10}, \ldots, C_{4\ell'-2})\\&+R\, \mathtt{poly}_{R^4}(C_3, C_7, C_{11}, \ldots, C_{4\ell'-1}).\end{aligned} \tag{5}$$

Then

$$\begin{aligned}&\mathtt{Poly1305}_R(C_0,\ldots,C_{\ell-1})\\&\quad = \begin{cases} P & \text{if } \rho = 0;\\ R\,\mathtt{poly}_R(P + C_{\ell-\rho}, C_{\ell-\rho+1},\ldots,C_{\ell-1}) & \text{if } \rho > 0.\end{cases}\end{aligned} \tag{6}$$

Define

$$\begin{aligned}&\boldsymbol{R} = \left(R^4, R^3, R^2, R\right)^{\mathsf{T}},\\&\boldsymbol{R}_4 = (R^4, R^4, R^4, R^4)^{\mathsf{T}},\\&\boldsymbol{C}_i = (C_{4i}, C_{4i+1}, C_{4i+2}, C_{4i+3})^{\mathsf{T}}, \quad \text{for } i = 0, 1, \ldots, \ell' - 1.\end{aligned} \tag{7}$$

The computation in (6) is described in vector form in Algorithm 1 (see Fig. 1). In the description of Algorithm 1 (Fig. 1), a temporary vector $\boldsymbol{T} = (T_0, T_1, T_2, T_3)$ is used and $\circ$ denotes the Hadamard (i.e. component-wise) product of vectors. The quantity P is a temporary field element.

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 the 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.

Let $(x_4, x_3, x_2, x_1, x_0)$ and $(y_4, y_3, y_2, y_1, y_0)$ be 5-limb representations 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:

$$\begin{aligned}z_0 &= x_0 \cdot y_0 + 5 \cdot x_1 \cdot y_4 + 5 \cdot x_2 \cdot y_3 + 5 \cdot x_3 \cdot y_2 + 5 \cdot x_4 \cdot y_1\\z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + 5 \cdot x_2 \cdot y_4 + 5 \cdot x_3 \cdot y_3 + 5 \cdot x_4 \cdot y_2\\z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + 5 \cdot x_3 \cdot y_4 + 5 \cdot x_4 \cdot y_3\\z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 + 5 \cdot x_4 \cdot y_4\\z_4 &= x_0 \cdot y_4 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 + x_4 \cdot y_0.\end{aligned} \tag{8}$$

Note that each z_i is less than 2^{64}. We define an operation `mult`, where the vector $(z_0,\ldots,z_4)$ denotes the output of $\mathtt{mult}((x_4,\ldots,x_0),(y_4,\ldots,y_0))$.

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 $\mathtt{reduce}(z_4,\ldots,z_0)$, we will denote the vector $(w_4,\ldots,w_0)$. For the details of the reduction step, we refer to [1].

`r0`	$5 \cdot r_4$	$5 \cdot r_3$	$5 \cdot r_2$	r_4	r_3	r_2	r_1	r_0
`r1`	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$

Fig. 2 *Packing of R^4 and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ into two 256-bit registers r_0 and r_1*

`t0`	$t_{3,2}$	$t_{3,0}$	$t_{2,2}$	$t_{2,0}$	$t_{1,2}$	$t_{1,0}$	$t_{0,2}$	$t_{0,0}$
`t1`	$t_{3,3}$	$t_{3,1}$	$t_{2,3}$	$t_{2,1}$	$t_{1,3}$	$t_{1,1}$	$t_{0,3}$	$t_{0,1}$
`t2`	x	$t_{3,4}$	x	$t_{2,4}$	x	$t_{1,4}$	x	$t_{0,4}$

Fig. 3 *Packing of four 5-limb vectors in three 256-bit registers* `t0`, `t1` *and* `t2`

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	$s_{4,3}$	$s_{4,2}$	$s_{4,1}$	$s_{4,0}$

Fig. 4 *Packing the output of* `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ *into five 256-bit registers* $S_0, \ldots, S_4$

Table 1 Names and counts of the various Intel intrinsic instructions required by `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ along with the latency and throughput figures on the Haswell and Skylake processors

Intrinsic	Count	(Latency, Throughput) Skylake	Haswell
_mm256_mul_epu32	25	(5, 0.5)	(5, 1)
_mm256_set_epi32	1	—	—
_mm256_add_epi64	20	(1, 0.33)	(1, 0.5)
_mm256_permutevar8 × 32_epi32	9	—	—
_mm256_permute4 × 64_epi64	4	—	—

`w0`	$w_{3,2}$	$w_{3,0}$	$w_{2,2}$	$w_{2,0}$	$w_{1,2}$	$w_{1,0}$	$w_{0,2}$	$w_{0,0}$
`w1`	$w_{3,3}$	$w_{3,1}$	$w_{2,3}$	$w_{2,1}$	$w_{1,3}$	$w_{1,1}$	$w_{0,3}$	$w_{0,1}$
`w2`	x	$w_{3,4}$	x	$w_{2,4}$	x	$w_{1,4}$	x	$w_{0,4}$

Fig. 5 *Packing the output of* `vecReduce`$(\boldsymbol{S})$ *into three 256-bit registers* `w0`, `w1`, `w2`

Suppose X is a fixed quantity and the product $X \cdot Y \bmod p$ is required to be computed. The computation in (8) 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 (Fig. 1) requires the vector multiplication

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

Note that the multiplication in Step 3 of Algorithm 1 (Fig. 1) has one of the operands to be fixed to $\boldsymbol{R}_4$ while the other operand changes. Goll and Gueron [5] presented a very efficient SIMD algorithm to perform this multiplication.

The vector $\boldsymbol{T} = (T_0, T_1, T_2, T_3)$ 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 twenty 26-bit quantities are required to store $\boldsymbol{T}$. 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 [5] uses 256-bit words. Each 256-bit word is considered to be eight 32-bit words. So, the twenty 26-bit quantities of $\boldsymbol{T}$ can be stored in three 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 two 256-bit words. The multiplication $\boldsymbol{W} = \boldsymbol{R}_4 \circ \boldsymbol{T}$ consists of two steps.

Vector multiplication step: This step takes as input the three 256-bit words representing $\boldsymbol{T} = (T_0, T_1, T_2, T_3)$ and the two 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 five 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 `mult`$((r_4, \ldots, r_0), (t_{j,4}, \ldots, t_{j,0}))$ for $j = 0, \ldots, 3$. Let $\boldsymbol{S} = (S_0, \ldots, S_4)$. By `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ we will denote $\boldsymbol{S}$.

Vector reduction step: This step takes as input $S_0, \ldots, S_4$ and produces as output three 256-bit words which store the twenty 26-bit (or 27-bit) words of the result. Let us call the result as $\boldsymbol{W}$. So, $\boldsymbol{W} = (W_0, W_1, W_2, W_3)$. Now let us define $W_j = (w_{j,4}, \ldots, w_{j,0})$, $j = 0, 1, 2, 3$. Then $(w_{j,4}, \ldots, w_{j,0})$ is the output of `reduce`$(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$. $\boldsymbol{W}$ will denote `vecReduce`(S).

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

We provide the top-level schematics of `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ and `vecReduce`$(\boldsymbol{S})$. 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 two 256-bit words, as shown in Fig. 2.

The vector$\boldsymbol{T} = (T_0, T_1, T_2, T_3)$ is stored in three 256-bit words, as shown in Fig. 3, where x denotes an undetermined quantity that is not used in the algorithm.

The Intel AVX2 implementation of `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$, uses a number of SIMD permutation operations on `t0`, `t1` and `t2` followed by 32-bit SIMD multiplication operations with `r0` and `r1` and 64-bit SIMD operations to accumulate the results. Finally, the result of `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ is $(S_0, \ldots, S_4)$ and is stored, as shown in Fig. 4.

The number of different Intel intrinsic instructions required by `vecMult`$(\boldsymbol{R}_4, \boldsymbol{T})$ is given in Table 1. The table also provides the latency and throughput figures in cycles for the Skylake and Haswell processors (These figures are available from the link https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (accessed on 12 February 2020). The corresponding figures for Kaby Lake are not available.)

The `vecReduce`$(\boldsymbol{S})$ implementation takes as input the five 256-bit words $S_0, \ldots, S_4$ and produces as output the vector $\boldsymbol{W} = (W_0, W_1, W_2, W_3)$ stored in three 256-bit words w_0, w_1 and w_2, as shown in Fig. 5.

The evaluation in Step 7 of Algorithm 1 is also done using vector operations. This is not clearly described in [5] 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.

3.2 Lazy reduction

Consider the loop in Steps 1–3 of Algorithm 1. Step 3 consists of one 4-way field multiplication $\boldsymbol{R}_4 \circ \boldsymbol{T}$ followed by a 4-way field addition. As explained above, the operation $\boldsymbol{R}_4 \circ \boldsymbol{T}$ can be realised as `vecReduce(vecMult(`$\boldsymbol{R}_4, \boldsymbol{T}$`))`. So, $\boldsymbol{R}_4 \circ \boldsymbol{T}$ requires one `vecMult` and one `vecReduce` operations. For long messages, it is possible to improve this by using a lazy reduction strategy. Such a strategy

```
    Input: $(D_0, \dots, D_{\ell-1})$
    Output: $\mathsf{Poly1305}_R(D_0, \dots, D_{\ell-1})$
1   $\mathbf{T} \leftarrow \mathbf{D}_0$;
2   for $i = 1$ to $m' - 1$ do
3       $\mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{D}_i$
4   $\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
5   return $T_0 + T_1 + T_2 + T_3$
```

Fig. 6 *Algorithm 2: Structure of the new 4-way vectorisation of* `Poly1305` *computation. Refer to (12) for the definition of the vector quantities*

consists of performing a series of successive `vecMult` and 4-way field additions followed by a single reduction. We provide more details.

Steps 1–3 perform $(\ell' - 1)$ 4-way field multiplications and 4-way field additions. Suppose we bunch these operations into groups where each group has λ 4-way field multiplications and λ 4-way field additions. If λ does not divide $\ell' - 1$, the last group may have a lesser number of such operations. With $\boldsymbol{T}$ initialised to $\boldsymbol{C}_0$, the computation of the jth group, $j = 0, \dots, \lfloor(\ell' - 1)/\lambda\rfloor$, processes $\boldsymbol{C}_{\lambda j+1}, \dots, \boldsymbol{C}_{\lambda j+\lambda}$. The actual computation is given as follows:

$$\begin{aligned} \boldsymbol{T} \leftarrow \boldsymbol{R}_{4\lambda} \circ \boldsymbol{T} + \boldsymbol{R}_{4(\lambda-1)} \circ \boldsymbol{C}_{\lambda j+1} \\ + \cdots + \boldsymbol{R}_4 \circ \boldsymbol{C}_{\lambda j+\lambda-1} + \boldsymbol{C}_{\lambda j+\lambda}. \end{aligned} \tag{9}$$

In the above, $\boldsymbol{R}_{4k} = (R^{4k}, R^{4k}, R^{4k}, R^{4k})^{\top}$ for $k = 1, \dots, \lambda$. For the multiplication by $\boldsymbol{R}_{4k}$, the field elements R^{4k} and $5 \cdot R^{4k}$ are precomputed and stored (only the first four limbs of $5 \cdot R^{4k}$ are stored).

As written, (9) requires λ 4-way field multiplications and λ 4-way field additions. Note however, that the results of the field multiplications are simply added together. This suggests the following lazy reduction strategy to perform the computation given in (9)

$$\begin{aligned} \boldsymbol{W}_0 &\leftarrow \texttt{vecMult}(\boldsymbol{R}_{4\lambda}, \boldsymbol{T}); \\ \boldsymbol{W}_k &\leftarrow \texttt{vecMult}(\boldsymbol{R}_{4k}, \boldsymbol{C}_{\lambda j+k}),\ k = 1, \dots, \lambda - 1; \\ \boldsymbol{B} &\leftarrow \boldsymbol{W}_0 + \cdots + \boldsymbol{W}_{\lambda-1} + \boldsymbol{C}_{\lambda j+\lambda}; \\ \boldsymbol{T} &\leftarrow \texttt{vecReduce}(\boldsymbol{B}). \end{aligned}$$

This method requires λ `vecMult` operations, λ 4-way field additions and a single `vecReduce` operation. Compared to a direct computation of (9), the lazy reduction strategy reduces the number of `vecReduce` operations by roughly a factor of $(\lambda - 1)/\lambda$. This would suggest that using a higher value of λ should always be beneficial. This, however, is not the case. As λ increases, so does the number of pre-computed quantities. The time for pre-computation has to be taken into account. For a higher value of λ not all the pre-computed quantities can be kept in the registers and as a result, the number of load/store operations would increase substantially. Also, adding too many of the products without a reduction can lead to an overfull in the register. These reasons prevent the use of high values of λ. In [5], the lazy reduction strategy was used for messages of lengths at least 832 bytes and with the values of λ to be 2 and 3.

4 New SIMD implementation of `Poly1305`

Algorithm 1 (Fig. 1) implements the computation in (5). If $4|\ell$ (i.e. $\rho = 0$), then the 4-way SIMD computation proceeds uniformly throughout. However, if $4\nmid\ell$, then the 4-way SIMD computation in Algorithm 1 (Fig. 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 $(C_0, \dots, C_{\ell-1})$ obtained as $\texttt{format}(M)$, define the sequence $(D_0, \dots, D_{m-1})$, where if $\rho = 0$, then $D_i = C_i$ for $i = 0, \dots, \ell - 1$ and if $\rho > 0$, then

$$D_i = \begin{cases} 0, & i = 0, \dots, 3 - \rho; \\ C_{i-4+\rho}, & i = 4 - \rho, \dots, m - 1. \end{cases}$$

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 $(C_0, \dots, C_{\ell-1})$ 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 $\texttt{Poly1305}_R(D_0, \dots, D_{m-1})$, we have

$$\texttt{Poly1305}_R(D_0, \dots, D_{m-1}) = \texttt{Poly1305}_R(C_0, \dots, C_{\ell-1}). \tag{10}$$

Let us define $m' = m/4$. Then, the computation of $\texttt{Poly1305}_R(D_0, \dots, D_{m-1})$ can be written as follows:

$$\begin{aligned} &\texttt{Poly1305}_R(D_0, \dots, D_{m-1}) \\ &\quad = R^4 \texttt{poly}_{R^4}(D_0, D_4, \dots, D_{m-4}) \\ &\quad + R^3 \texttt{poly}_{R^4}(D_1, D_5, \dots, D_{m-3}) \\ &\quad + R^2 \texttt{poly}_{R^4}(D_2, D_6, \dots, D_{m-2}) \\ &\quad + R^1 \texttt{poly}_{R^4}(D_3, D_7, \dots, D_{m-1}). \end{aligned} \tag{11}$$

In a manner similar to (7), define

$$\boldsymbol{D}_i = (D_{4i}, D_{4i+1}, D_{4i+2}, D_{4i+3})^{\top}; \quad \text{for } i = 0, \dots, m' - 1. \tag{12}$$

The computation in (11) is described in vector form in Algorithm 2 (see Fig. 6).

In Algorithm 2 (Fig. 6), 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 $\texttt{vecMult}(\cdot, \cdot)$ followed by $\texttt{vecReduce}(\cdot)$.

For the case $\rho = 0$, Step 7 of Algorithm 1 (Fig. 1) is not executed. In this case, Algorithms 1 and 2 (Figs. 1 and 6) become the same. For $\rho = 3$, there is a performance improvement of Algorithm 2 (Fig. 6) over Algorithm 1 (Fig. 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 (Fig. 6) does not necessarily lead to speed gains. We address this issue in the next section.

Remark: We have described Algorithm 2 (Fig. 6) 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 (Fig. 6), the first multiplication is $\boldsymbol{R}_4 \circ \boldsymbol{D}_0$. Consider $\boldsymbol{D}_0 = (D_0, D_1, D_2, D_3)^{\top}$. Depending on the value of ρ, there are four cases.

$$\boldsymbol{D}_0 = \begin{cases} (C_0, C_1, C_2, C_3)^{\top} & \text{if } \rho = 0; \\ (0, 0, 0, C_0)^{\top} & \text{if } \rho = 1; \\ (0, 0, C_0, C_1)^{\top} & \text{if } \rho = 2; \\ (0, C_0, C_1, C_2)^{\top} & \text{if } \rho = 3. \end{cases} \tag{13}$$

Suppose $\rho = 1$ so that $\boldsymbol{D}_0 = (0, 0, 0, C_0)$. Let the 5-limb representation of C_0 be given by $(c_{0,4}, \dots, c_{0,0})$. Consider the schematic of the operation `vecMult` as discussed in Section 3.1. The representation of $\boldsymbol{D}_0$ in the three 256-bit words $\mathtt{t}_0, \mathtt{t}_1$ and $\mathtt{t}_2$ will look as shown in Fig. 7 (where x is a do not care value).

Since a lot of entries in the above representation are zeros, applying the Goll–Gueron `vecMult` operation to $\boldsymbol{R}_4$ and this $\boldsymbol{D}_0$

t0	$c_{0,2}$	$c_{0,0}$	0	0	0	0	0	0
t1	$c_{0,3}$	$c_{0,1}$	0	0	0	0	0	0
t2	x	$c_{0,4}$	x	0	x	0	x	0

Fig. 7 *Packing of the input for* $\rho = 1$ *if the Goll–Gueron strategy is followed for the first vector multiplication, i.e. for the initial multiplication*

t0	0	$c_{0,3}$	0	$c_{0,2}$	0	$c_{0,1}$	0	$c_{0,0}$
t1	0	$c_{0,4}$	0	0	0	0	0	0

Fig. 8 *Packing of the input used for the initial multiplication for* $\rho = 1$ *by the new algorithm*

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

Fig. 9 *Packing the output of the multiplication for* $\rho = 1$ *into five 256-bit registers* $S_0, \ldots, S_4$

t0	$c_{0,3}$	$c_{1,3}$	$c_{0,2}$	$c_{1,2}$	$c_{0,1}$	$c_{1,1}$	$c_{0,0}$	$c_{1,0}$
t1	0	$c_{1,4}$	0	$c_{0,4}$	0	0	0	0

Fig. 10 *Packing of the input used for the initial multiplication for* $\rho = 2$ *by the new algorithm*

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

Fig. 11 *Packing the output of the multiplication for* $\rho = 2$ *into five 256-bit registers* $S_0, \ldots, S_4$

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 $\boldsymbol{D}_0$ 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 $\boldsymbol{D}_0 = (0, 0, C_0, C_1)$. When $\rho = 3$, $\boldsymbol{D}_0 = (0, C_0, C_1, C_2)$, 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 $\boldsymbol{D}_0$ *for the case* $\rho = 1$*:* In this case, $\boldsymbol{D}_0 = (0, 0, 0, C_0)$ is represented using two 256-bit words, as shown in Fig. 8. The five 256-bit words holding the output of `vecMult`$(\boldsymbol{R}_4, \boldsymbol{D}_0)$ in this case will have the form, as shown in Fig. 9.

Representation of $\boldsymbol{D}_0$ *for the case* $\rho = 2$*:* In this case, $\boldsymbol{D}_0 = (0, 0, C_0, C_1)$ is represented using two 256-bit words, as shown in Fig. 10. The five 256-bit words holding the output of `vecMult`$(\boldsymbol{R}_4, \boldsymbol{D}_0)$ in this case will have the form, as shown in Fig. 11.

In both the cases, the representation of $\boldsymbol{R}_4$ using two 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 $\boldsymbol{S}$ in five 256-bit words as shown above. The `vecReduce` algorithm mentioned in Section 3.1 is applied to $\boldsymbol{S}$ to obtain the result $\boldsymbol{R}_4 \circ \boldsymbol{D}_0$. The output of `vecReduce` is in the form of three 256-bit words which is stored in $\boldsymbol{T}$. The further multiplications $\boldsymbol{R}_4 \circ \boldsymbol{T}$ in the loop at Step 4 of Algorithm 2 (Fig. 6) 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.

4.2 Lazy reduction

The lazy reduction strategy described in Section 3.2 applies to the loop in Steps 1–3 of Algorithm 2 (Fig. 6). The computation is divided into groups where each group processes λ of the $\boldsymbol{D}_i$'s. As in the case of Algorithm 1 (Fig. 1), the lazy reduction strategy requires only a $1/\lambda$ fraction of the number of reductions required in a direct implementation of Algorithm 2 (Fig. 6). Following the code for [5], we have incorporated the lazy reduction strategy for messages having at least 832 bytes with values of λ to be 2 or 3.

Table 2 Summary of the comparative performance analysis of the new code with the code of [5] using the gcc compiler

Processor	Range	Max speed-up, %	Avg speed-up, %
Haswell	49B–1000B	29.70	12.06
	1001B–2000B	22.31	12.46
	2001B–4000B	15.15	9.06
Skylake	49B–1000B	36.81	15.05
	1001B–2000B	15.24	6.94
	2001B–4000B	12.66	3.29
Kaby Lake	49B–1000B	35.33	13.12
	1001B–2000B	21.49	12.94
	2001B–4000B	21.17	10.51

Table 3 Summary of the comparative performance analysis of the new code with the code of [5] using the clang compiler

Haswell	49B–1000B	23.33%	12.92%
	1001B–2000B	14.52%	5.27%
	2001B–4000B	10.20%	2.44%
Skylake	49B–1000B	29.61%	12.95%
	1001B–2000B	21.49%	8.99%
	2001B–4000B	20.45%	5.60%
Kaby Lake	49B–1000B	29.43%	10.57%
	1001B–2000B	18.58%	4.54%
	2001B–4000B	10.84%	1.64%

5 Implementation and comparison

We have implemented the SIMD strategy given in Algorithm 2 (Fig. 6) for the evaluation of the Poly1305 hash function. This implementation consisted of modifying the Intel intrinsic code implementing the SIMD strategy in [5]. Portions of the code were used without any change. In particular, the basic 4-way multiplication routine of [5] has been directly used. On the other hand, the improved initial multiplication algorithms are new to our SIMD strategy and had to be implemented. The modified code is publicly available at the following link: https://github.com/Sreyosi/Improved-SIMD-Implementation-of-Poly1305.

The performance has been measured in terms of the number of machine cycles per byte under the same conditions as mentioned in [5]: Intel Turbo Boost Technology, Intel Hyper-Threading Technology and Intel Speedstep Technology were disabled. For a comparative study, performances of the new code and that of the code accompanying [5] were measured using the same strategy and on the same computers.

Measurements were made on the following platforms:

- *Haswell:* Intel Core i7-4790 CPU @ 3.60 GHz x 8; running Ubuntu 18.04.2 LTS (64-bit); gcc version 7.4.0; Clang version 8.4.
- *Skylake:* Intel Core i7-6500U CPU @ 2.50 GHz x 2; running Ubuntu 14.04 LTS (64-bit); gcc version 5.5.0; Clang version 8.4.
- *Kaby Lake:* Intel Core i7-7700U CPU @ 3.60 GHz x 4; running Ubuntu 18.04 LTS (64-bit); gcc version 7.3.0; Clang version 8.4.

In all cases, measurements were made on a single core of the specified machines. The compile commands used are as follows:

- `gcc -mavx2 -O3 -fomit-frame-pointer`,
- `clang -mavx2 -O2 -fomit-frame-pointer`.

We note that in our experiments, we have found that for the clang compiler, in general, the-O2 option yields a faster code than the-O3 option. So, we have carried out all measurements using the-O2 option.

Message length: For measuring performance and comparison to [5], we considered messages with lengths up to 4KB (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 27 June 2019.). 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 in 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. Suppose that for a message length l bytes, the Goll–Gueron code requires t_0 cycles/byte and the new code requires t_1 cycles/byte. Then the speed-up (in percentage) attained for message length l is $\mathtt{su}_l = 100(t_1 - t_0)/t_0$. The average speed-up is the average of all the $\mathtt{su}_l$'s.

A top-level summary of the comparison using gcc compiler is as follows:

- *Haswell:* speed-up has been obtained in 99.87% cases of the message lengths that were considered; in 0.07% cases, the performances of both the codes were the same; in 0.05% cases, the new code has shown a slight slowdown.
- *Skylake:* speed-up has been obtained in 89.51% cases of the message lengths that were considered; in 6.27% cases, the performances of both the codes were the same; in 4.21% cases, the new code has shown a slight slowdown.
- *Kaby Lake:* speed-up has been obtained in 99.66% cases of the message lengths that were considered; in 0.17% cases, the performances of both the codes were the same; in 0.15% cases, the new code has shown a slight slowdown.

Tables 2 and 3 provide a summary of the maximum and average speed-ups for the three platforms for three different ranges of message lengths. Table 2 has been obtained using the gcc compiler, while Table 3 has been obtained using the clang compiler. The tables show that for short messages, on an average of about 10–15% speed-up is obtained.

The actual values of cycles/byte that were obtained for the new code and the code of [5] for 12 different message lengths are shown in Tables 4–6. Measurements for both clang and gcc are provided. The message lengths were chosen to show variations in the improvements obtained by the new code over the code of [5].

Providing measurements for all lengths up to 4000 bytes using tables is cumbersome. So, we provide plots of such data. Two kinds of plots are provided for each of the processors. The first kind plots the speed-up with the size of the message, while the second kind plots the values of cycles/byte with the message size.

Table 4 Performance on Haswell using clang and gcc

Length	clang			gcc		
	Goll–Gueron	New	Speed-up, %	Goll–Gueron	New	Speed-up, %
56	4.71	4.36	7.43	4.71	4.36	7.43
80	3.85	4.00	3.75	3.70	3.50	5.40
96	3.33	3.25	2.40	3.17	2.96	6.62
112	3.02	2.61	13.57	3.11	2.64	15.11
160	3.32	2.96	10.84	2.30	2.17	5.62
224	2.00	1.88	6.00	1.88	1.79	4.78
240	2.08	1.78	14.42	1.93	1.70	11.92
496	1.43	1.29	9.79	1.37	1.24	9.48
600	1.37	1.21	11.67	1.31	1.21	7.63
800	1.14	1.00	12.28	1.09	1.11	−1.83
1000	1.26	1.07	15.07	1.28	1.17	8.59
2000	0.92	0.92	0	0.94	0.89	5.31

Table 5 Performance on Skylake using clang and gcc

Length	clang			gcc		
	Goll–Gueron	New	Speed-up, %	Goll–Gueron	New	Speed-up, %
56	4.57	4.11	10.06	4.39	3.79	13.66
80	3.48	3.40	2.29	3.33	3.23	3.00
96	2.98	2.92	2.01	2.77	2.75	0.72
112	3.02	2.61	13.57	2.80	2.43	13.21
160	2.17	2.09	3.68	2.05	1.94	5.36
224	2.92	2.08	28.76	1.66	1.61	3.01
240	1.90	1.62	14.73	1.76	1.56	11.36
496	1.26	1.14	9.52	1.21	1.14	5.78
600	1.20	1.07	10.83	1.19	1.08	9.24
800	0.99	0.95	4.04	0.97	0.98	-1.03
1000	1.03	0.93	9.70	1.01	0.92	8.91
2000	0.78	0.77	1.28	0.76	0.78	-2.63

Table 6 Performance on Kaby Lake using clang and gcc

Length	clang			gcc		
	Goll–Gueron	New	Speed-up, %	Goll–Gueron	New	Speed-up, %
56	4.57	4.11	10.06	4.43	3.79	14.44
80	3.55	3.50	1.40	3.27	3.15	3.66
96	2.94	2.90	1.36	2.79	2.58	7.52
112	2.98	2.64	11.04	2.86	2.27	20.62
160	2.17	2.09	3.68	2.05	1.90	7.31
224	1.78	1.67	6.17	1.64	1.54	6.09
240	1.85	1.61	12.97	1.75	1.44	17.71
496	1.21	1.12	7.43	1.2	1.1	8.33
600	1.17	1.08	7.69	1.16	1.05	9.48
800	0.94	0.97	−3.19	0.94	0.97	−3.19
1000	1.04	0.93	11.00	1.02	0.93	8.82
2000	0.78	0.77	1.28	0.77	0.77	0

These plots are provided only for the gcc compiler. The plots for the clang compiler are similar and so we do not provide such plots. For Haswell, Fig. 12 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 Figs. 13–15. For Skylake, Fig. 16 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 Figs. 17–19. For Kaby Lake, Fig. 20 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 Figs. 21–23.

Comparison on other processors: The comparative measurements show the efficiency gain obtained by the new code over the code of [5]. Due to the availability of computational resources, we have been able to perform measurements only on three Intel processors. The efficiency improvements, though, are a result of modifying the algorithm for computing `Poly1305` to better fit vectorised implementations. So, other processors that support vector implementations should also profit from the new algorithm.

Alternative implementations of `Poly1305`*:* A reviewer of the paper has pointed out that [7, 8] provide alternative implementations of `Poly1305`. The implementation in [7] is faster than the implementation in [8], so, we consider only [7].

It is mentioned in [7] that the paper utilises 'the same optimisation ideas used by the best implementations', namely those adopted in OpenSSL and in work by Goll and Gueron [5]. The implementation strategy in [7] is the following. For messages of lengths up to 256 bytes, a 64-bit implementation is used. For messages of lengths greater than 256 bytes, vector implementation

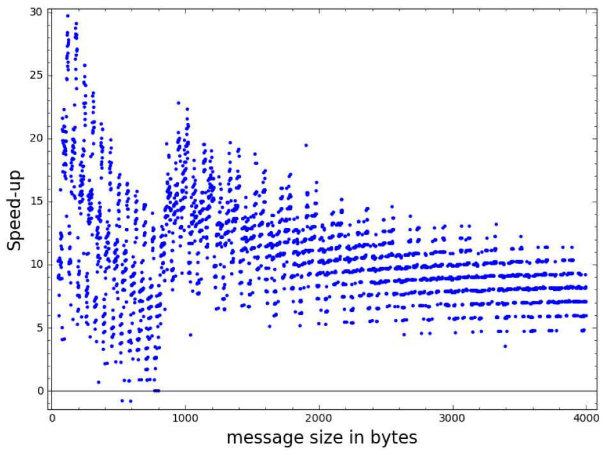

Fig. 12 *Speed-up (%) versus message (bytes) for gcc and Haswell*

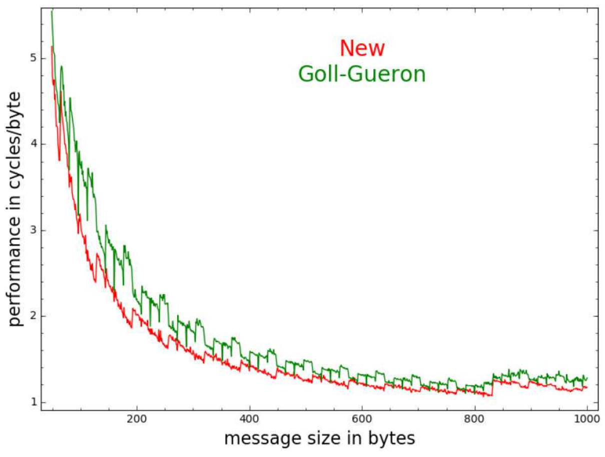

Fig. 13 *cycles/byte versus message size (49–1000 bytes) for gcc and Haswell*

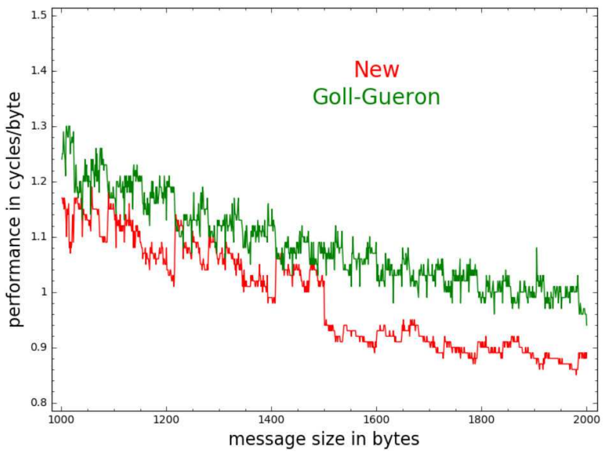

Fig. 14 *cycles/byte versus message size (1001–2000 bytes) for gcc and Haswell*

is used in a manner that is similar to that of the Goll–Gueron strategy. Suppose there are ℓ blocks, and as before let $\rho = \ell \bmod 4$ and $\ell' = (\ell - \rho)/4$. The vector evaluation is done four blocks at a time up to ℓ' blocks. Then the result is reformatted into 64-bit words and the remaining ρ blocks are processed using 64-bit arithmetic in a manner similar to the processing of messages of lengths up to 256 bytes.

The implementation strategy that we suggest, namely to balance out the 4-way SIMD execution by pre-pending some zero blocks, is not present in [7]. If this strategy is adopted into the implementation of [7], then the reformatting into 64-bit words and processing the left-over message consisting of ρ blocks using 64-bit arithmetic will not be required.

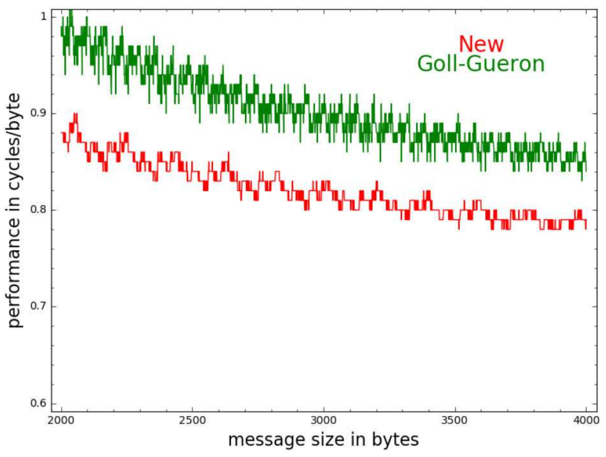

Fig. 15 *cycles/byte versus message size (2001–4000 bytes) graph for gcc and Haswell*

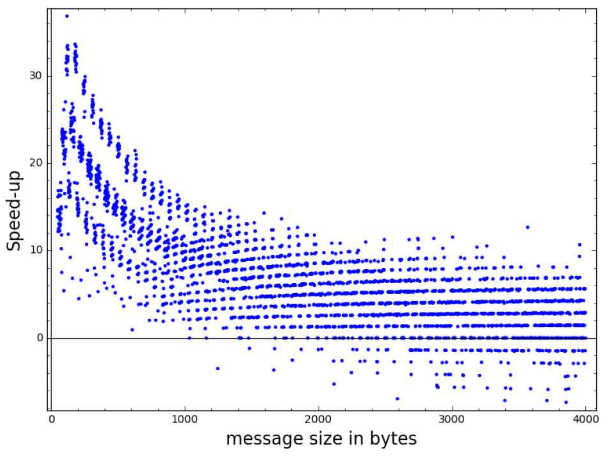

Fig. 16 *Speed-up (%) versus message size (bytes) graph for gcc and Skylake*

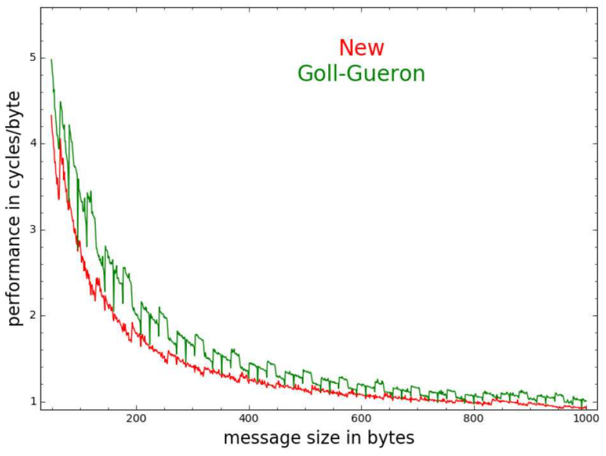

Fig. 17 *cycles/byte versus message size (49–1000 bytes) graph for gcc and Skylake*

The code for [7] has been written in the Jasmin language and the Jasmin compiler has been used to generate the assembly, which, according to the authors, is ‘as efficient as a hand-written assembly’. The code that we use, on the other hand, is in Intel intrinsic and compiled using gcc/clang. In general, intrinsic compiled code is slower than hand-written assembly and this holds for the comparison between the Jasmin compiled code and our code. A possible future work would be to code the strategy for balancing 4-way SIMD that we suggest into Jasmin and then perform a comparison with the code of [7].

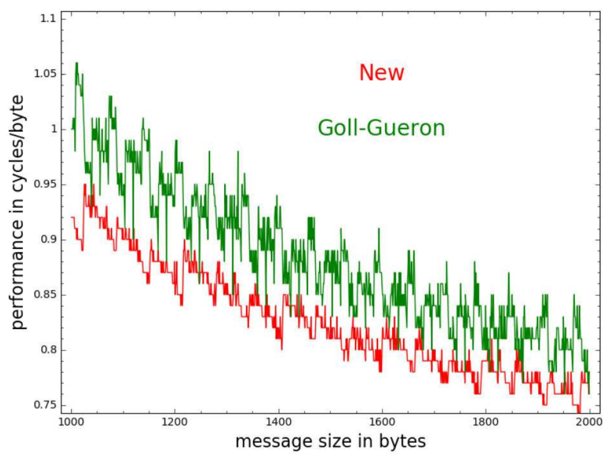

Fig. 18 *cycles/byte versus message size (1001–2000 bytes) graph for gcc and Skylake*

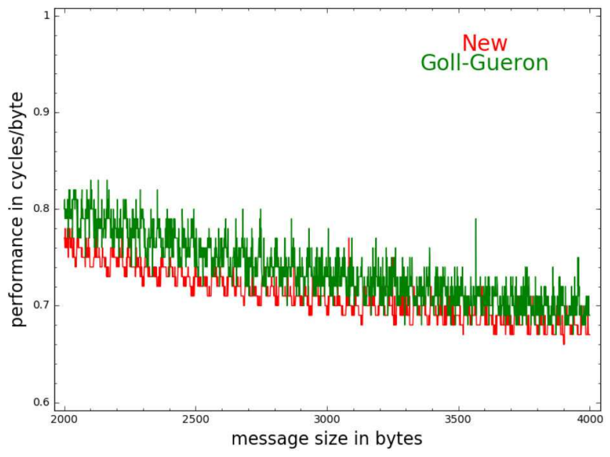

Fig. 19 *cycles/byte versus message size (2001–4000 bytes) graph for gcc and Skylake*

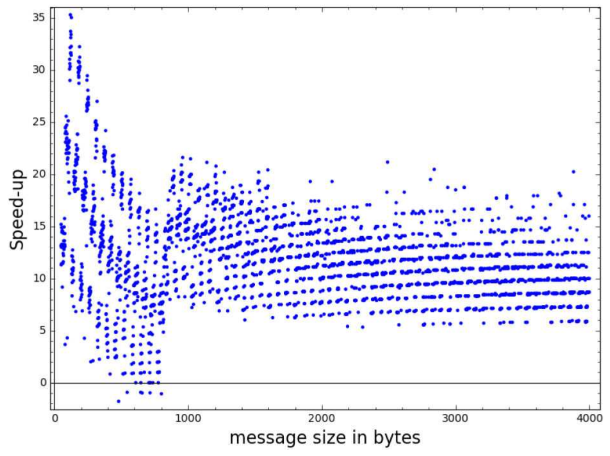

Fig. 20 *Speed-up (%) versus message size (bytes) graph for gcc and Kaby Lake*

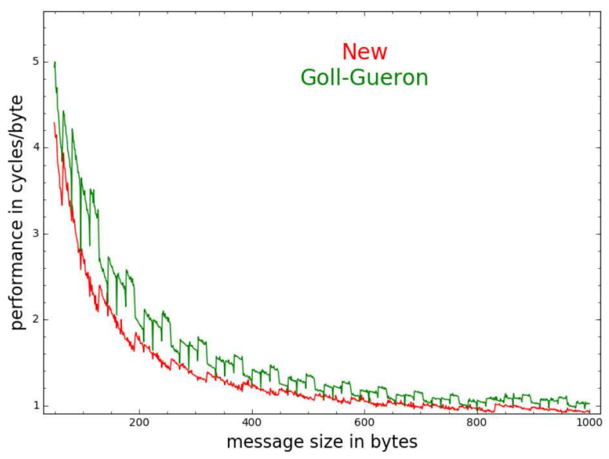

Fig. 21 *cycles/byte versus message size (49–1000 bytes) graph for gcc and Kaby Lake*

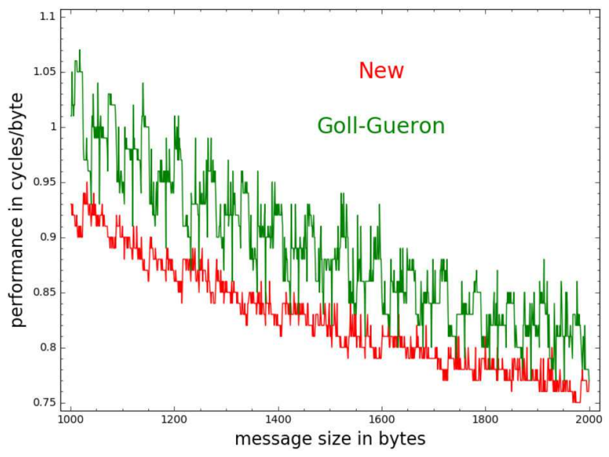

Fig. 22 *cycles/byte versus message size (1001–2000 bytes) graph for gcc and Kaby Lake*

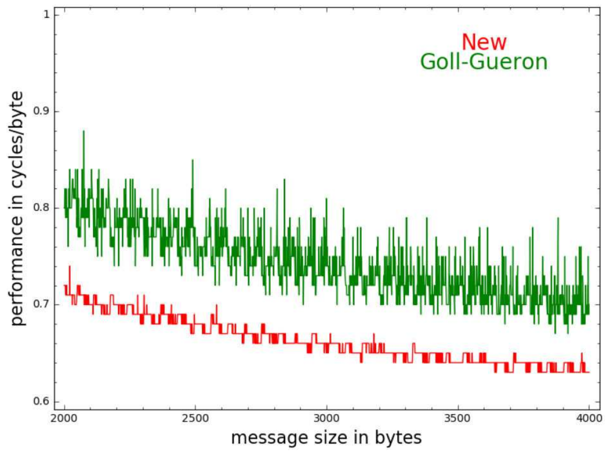

Fig. 23 *cycles/byte versus message size (2001–4000 bytes) graph for gcc and Kaby Lake*

6 Conclusion

In this work, we have proposed a simple modification to the previous Goll–Gueron strategy for SIMD implementations 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 four.

7 Acknowledgment

The authors thank Shay Gueron for kindly sharing the code associated with [5] with them and for providing comments on an earlier version of this paper. They also thank the reviewers for their kind comments, which have helped in improving the paper.

8 References

[1] Bernstein, D.J.: 'The Poly1305-AES-aes message-authentication code'. Fast Software Encryption: 12th Int. Workshop, FSE 2005, Paris, France, 21–23 February 2005, Revised Selected Papers, (LNCS, **3557**), pp. 32–49

[2] Bernstein, D.J.: 'Chacha, a variant of Salsa20'. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, Lausanne, Switzerland, January 2008. Available at http://cr.yp.to/chacha/chacha-20080128.pdf

[3] Bernstein, D.J., Lange, T., Schwabe, P.: 'The security impact of a new cryptographic library'. Progress in Cryptology – LATINCRYPT 2012 – 2nd

Int. Conf. on Cryptology and Information Security in Latin America, Santiago, Chile, 7–10 October 2012 (LNCS, **7533**), pp. 159–176

[4] Bernstein, D.J.: 'The Salsa20 family of stream ciphers'. Available at http://cr.yp.to/papers.html#salsafamily. Document ID: 31364286077dcdff8e4509f9ff3139ad. Date: 2007.12.25

[5] Goll, M., Gueron, S.: 'Vectorization of Poly1305 message authentication code'. 2015 12th Int. Conf. on Information Technology – New Generations, Las Vegas, NV, USA, April 2015, pp. 145–150, doi: 10.1109/ITNG.2015.28

[6] Chakraborty, D., Ghosh, S., Sarkar, P.: 'A fast single-key two-level universal hash function', *IACR Trans. Symmetric Cryptol.*, 2017, **2017**, (1), pp. 106–128

[7] Almeida, J.B., Barbosa, M., Barthe, G., *et al.*: 'The last mile: high-assurance and high-speed cryptographic implementations'. CoRR, abs/1904.04606, 2019

[8] Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., *et al.*: 'Implementing and proving the TLS 1.3 record layer'. 2017 IEEE Symp. on Security and Privacy, SP 2017, San Jose, CA, USA, 22–26 May 2017, pp. 463–482