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}

October 1, 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 page¹ 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 Bernstein².

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

²https://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.

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 \( \ell \) 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 \( \ell \) 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 \( \text{format}(M) \) 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.

\[
\text{Poly1305}_R(M) = C_0 R^L + C_1 R^{L-1} + \cdots + C_{\ell-2} R^2 + C_{\ell-1} R
\]

(1)

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

\[
\text{poly}_R(C_0, \ldots, C_{\ell-1}) = C_0 R^{\ell-1} + \cdots + C_{\ell-2} R + C_{\ell-1}.
\]

(2)

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

The definition of \( \text{poly} \) in (2) permits the computation of the output using Horner’s rule in the following manner.

\[
\text{poly}_R(C_0, \ldots, C_{\ell-1}) = ((\cdots(((C_0 R + C_1) R + C_2) R + \cdots) R + C_{\ell-1}.
\]

(3)

This requires \( \ell - 1 \) multiplications and \( \ell - 1 \) additions over \( \mathbb{F}_p \). As a result, \( \text{Poly1305} \) can be computed using \( \ell \) 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 \( \rho = \ell \mod 4 \) and \( \ell' = (\ell - \rho)/4 \). The computation of \( \text{Poly1305}_R(C_0, \ldots, C_{\ell-1}) \) can be done in the following manner. Let

\[
P = R^4 \text{poly}_R(C_0, C_4, C_8, \ldots, C_{4\ell'-4}) \\
+ R^3 \text{poly}_R(C_1, C_5, C_9, \ldots, C_{4\ell'-3}) \\
+ R^2 \text{poly}_R(C_2, C_6, C_{10}, \ldots, C_{4\ell'-2}) \\
+ R \text{poly}_R(C_3, C_7, C_{11}, \ldots, C_{4\ell'-1})
\]

(4)

Then

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

(5)

Define

\[
\textbf{R} = (R^4, R^3, R^2, R)^T; \\
\textbf{R}_4 = (R^4, R^3, R^2, R^1)^T; \\
\textbf{C}_i = (C_{4i}, C_{4i+1}, C_{4i+2}, C_{4i+3})^T \text{ for } i = 0, 1, \ldots, \ell' - 1.
\]

(6)
Here the subscript \( (\cdot)^T \) 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 \( T = (T_0, T_1, T_2, T_3) \) is used and \( \odot \) denotes the Hadamard (i.e., component-wise) product of vectors. The quantity \( P \) is a temporary field element.

\[
\text{Algorithm 1: Structure of Goll-Gueron 4-way vectorisation of Poly1305 computation.}
\]\begin{algorithm}
\begin{small}
Input: \( (C_0, \ldots, C_{\ell-1}) \)
\end{small}
\begin{small}
Output: Poly1305_\(R\)(\(C_0, \ldots, C_{\ell-1}\))
\end{small}
\begin{algorithmic}
1 \hspace{1em} \( T \leftarrow C_0 \);
2 \hspace{1em} for \( i = 1 \) to \( \ell' - 1 \) do
3 \hspace{2em} \( T \leftarrow R \odot T + C_i \)
4 \hspace{1em} \( T \leftarrow R \odot T \)
5 \hspace{1em} \( P = T_0 + T_1 + T_2 + T_3 \)
6 \hspace{1em} if \( \rho > 0 \) then
7 \hspace{2em} \( P \leftarrow R \text{poly} \_\(P + C_{\ell-\rho}, C_{\ell-\rho+1}, \ldots, C_{\ell-1}\) \)
8 \hspace{1em} return \( P \)
\end{algorithmic}
\end{algorithm}

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 \( \theta \) 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 \( X \cdot Y \mod p \) is computed in two steps.

\textit{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.

\[
\begin{align*}
z_0 &= x_0 \cdot y_0 + 5 \cdot x_1 \cdot y_4 + 5 \cdot x_2 \cdot y_3 + 5 \cdot x_3 \cdot y_2 + 5 \cdot x_4 \cdot y_1 \\
z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + 5 \cdot x_2 \cdot y_4 + 5 \cdot x_3 \cdot y_3 + 5 \cdot x_4 \cdot y_2 \\
z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + 5 \cdot x_3 \cdot y_4 + 5 \cdot x_4 \cdot y_3 \\
z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 + 5 \cdot x_4 \cdot y_4 \\
z_4 &= x_0 \cdot y_4 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 + x_4 \cdot y_0.
\end{align*}
\]

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

\textit{Reduction step:} \( W = w_0 + w_1\theta + \cdots + w_4\theta^4 \) is obtained such that \( W \equiv Z \mod 4 \) and each \( w_i \) can be represented using either 26 or 27 bits. By \text{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 \( X \cdot Y \mod p \) 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

$$R_4 \cdot T = (R^4 \cdot T_0, R^4 \cdot T_1, R^4 \cdot T_2, R^4 \cdot T_3)^T.$$  

Note that the multiplication in Step 3 of Algorithm 1 has one of the operands to be fixed to $R_4$ while the other operand changes. Goll and Gueron [6] presented a very efficient SIMD algorithm to do this multiplication.

The vector $T = (T_0, T_1, T_2, T_3)$ 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 $T$. Since intermediate results are not fully reduced, some of the $t_{i,j}$’s can be 27-bit quantities. Let $(r_{4,0}, \ldots, r_{4})$ 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 $T$ can be stored in 3 256-bit words. The vectors $(r_{1,4}, \ldots, r_{1})$ 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 $W = R_4 \cdot T$ consists of two steps.

Vector multiplication step: This step takes as input the 3 256-bit words representing $T = (T_0, T_1, T_2, T_3)$ and the 2 256-bit words representing $(r_{1,4}, \ldots, r_{1})$ 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 $\text{mult}((r_{4,0}, \ldots, r_{4}), (t_{j,4}, \ldots, t_{j,0}))$ for $j = 0, 1, 2, 3$. Let $S = (S_0, \ldots, S_4)$. By $\text{vecMult}(R_4, T)$ we will denote $S$.

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 $W = (W_0, W_1, W_2, W_3)$. 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 $\text{reduce}(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$. By $\text{vecReduce}(S)$ we will denote $W$.

In terms of the above notation, the computation $W = R_4 \cdot T$ consists of the following two steps: $S \leftarrow \text{vecMult}(R_4, T); W \leftarrow \text{vecReduce}(S)$. Note that $T$ is stored in 3 256-bit words and the output $W$ is also stored in 3 256-bit words. This ensures that the same multiplication algorithm can be applied to multiply $R_4$ and $W$, and so on.

We provide the top level schematics of $\text{vecMult}(R_4, T)$ and $\text{vecReduce}(S)$. The 5-limb representation $(r_{4,3}, r_{2,3}, r_{1,3})$ 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.

$$\begin{array}{ccccccccc}
\text{r}_0 & 5 \cdot r_{4} & 5 \cdot r_{3} & 5 \cdot r_{2} & r_{4} & r_{3} & r_{2} & r_{1} & r_{0} \\
\text{r}_1 & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} & 5 \cdot r_{1} \\
\end{array}$$

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

$$\begin{array}{cccccc}
\text{t}_0 & t_{3,2} & t_{3,0} & t_{2,2} & t_{2,0} & t_{1,2} & t_{1,0} & t_{0,2} & t_{0,0} \\
\text{t}_1 & t_{3,3} & t_{3,1} & t_{2,3} & t_{2,1} & t_{1,3} & t_{1,1} & t_{0,3} & t_{0,1} \\
\text{t}_2 & x & t_{3,4} & x & t_{2,4} & x & t_{1,4} & x & t_{0,4} \\
\end{array}$$

The Intel AVX2 implementation of $\text{vecMult}(R_4, T)$, uses a number of SIMD permutation operations on $t_0, t_1$ and $t_2$ followed by 32-bit SIMD multiplication operations with $r_0$ and $r_1$ and 64-bit SIMD operations to accumulate the results. Finally, the result of $\text{vecMult}(R_4, T)$ is $(S_0, \ldots, S_4)$ and is stored as follows.
The actual computation is the following.

\[
\begin{align*}
S_0 & = s_{0,3} \quad s_{0,2} \quad s_{0,1} \quad s_{0,0} \\
S_1 & = s_{1,3} \quad s_{1,2} \quad s_{1,1} \quad s_{1,0} \\
S_2 & = s_{2,3} \quad s_{2,2} \quad s_{2,1} \quad s_{2,0} \\
S_3 & = s_{3,3} \quad s_{3,2} \quad s_{3,1} \quad s_{3,0} \\
S_4 & = s_{4,3} \quad s_{4,2} \quad s_{4,1} \quad s_{4,0}
\end{align*}
\]

The number of different operations (in Intel intrinsics) required by \( \text{vecMult}(R_4, T) \) is given in Table 1.

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

\[
\begin{align*}
w_0 & = w_{3,2} \quad w_{3,0} \quad w_{2,2} \quad w_{2,0} \quad w_{1,2} \quad w_{1,1} \quad w_{0,2} \quad w_{0,0} \\
w_1 & = w_{3,3} \quad w_{3,1} \quad w_{2,3} \quad w_{2,1} \quad w_{1,3} \quad w_{1,1} \quad w_{0,3} \quad w_{0,1} \\
w_2 & = x \quad w_{3,4} \quad x \quad w_{2,4} \quad x \quad w_{1,4} \quad x \quad w_{0,4}
\end{align*}
\]

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.

### 3.2 Lazy Reduction

Consider the loop in Steps 1-3 of Algorithm 1. Step 3 consists of one 4-way field multiplication \( R_4 \circ T \) followed by a 4-way field addition. As explained above, the operation \( R_4 \circ T \) can be realised as \( \text{vecReduce}(\text{vecMult}(R_4, T)) \). So, \( R_4 \circ T \) requires one \( \text{vecMult} \) and one \( \text{vecReduce} \) operation. For long messages, it is possible to improve this by using a lazy reduction strategy. Such a strategy consists of performing a series of successive \( \text{vecMult} \) and 4-way field additions followed by a single reduction. We provide more details.

Steps 1-3 has \( (\ell' - 1) \) 4-way field field multiplications and 4-way field additions. Suppose we bunch these operations into groups where each group has \( \lambda \) 4-way field multiplications and \( \lambda \) 4-way field additions. If \( \lambda \) does not divide \( \ell' - 1 \), the last group may have lesser number of such operations. With \( T \) initialised to \( C_0 \), the computation of the \( j \)-th group, \( j = 0, \ldots, \lfloor (\ell' - 1)/\lambda \rfloor \), processes \( C_{\lambda j+1}, \ldots, C_{\lambda j+\lambda} \).

The actual computation is the following.

\[
T \leftarrow R_{4\lambda} \circ T + R_{4(\lambda-1)} \circ C_{\lambda j+1} + \cdots + R_4 \circ C_{\lambda j+\lambda-1} + C_{\lambda j+\lambda}.
\]  

(8)

In the above, \( R_{4k} = (R^{4k}, R^{4k}, R^{4k}, R^{4k})^T \) for \( k = 1, \ldots, \lambda \). For the multiplication by \( R_{4k} \), the field elements \( R^{4k} \) and \( 5 \cdot R^{4k} \) are precomputed and stored (only the first 4 limbs of \( 5 \cdot R^{4k} \) are stored).

As written, (8) requires \( \lambda \) 4-way field multiplications and \( \lambda \) 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 (8).
\[ W_0 \leftarrow \text{vecMult}(R_{4\lambda}, T); \]
\[ W_k \leftarrow \text{vecMult}(R_{4k}, C_{\lambda j+k}), k = 1, \ldots, \lambda - 1; \]
\[ B \leftarrow W_0 + \cdots + W_{\lambda - 1} + C_{\lambda j+\lambda}; \]
\[ T \leftarrow \text{vecReduce}(B). \]

This method requires \( \lambda \) \text{vecMult} operations, \( \lambda \) 4-way field additions and a single \text{vecReduce} operation. Compared to a direct computation of (8), the lazy reduction strategy reduces the number of \text{vecReduce} operations by roughly a factor of \((\lambda - 1)/\lambda\). This would suggest that using a higher value of \( \lambda \) should always be beneficial. This, however, is not the case. As \( \lambda \) 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 \( \lambda \) 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 overflow in the register. These reasons prevent the use of high values of \( \lambda \). In [6], the lazy reduction strategy was used for messages of lengths at least 832 bytes and with the values of \( \lambda \) to be 2 and 3.

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 \( 4 \nmid \ell \), then the 4-way SIMD computation in Algorithm 1 proceeds uniformly for \( \ell' \) 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 \( \rho = \ell \mod 4 \). 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 \text{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

\[
\begin{align*}
D_i &= 0 & \text{for } i = 0, \ldots, 3 - \rho; \\
D_i &= C_{i-4+\rho} & \text{for } i = 4 - \rho, \ldots, m - 1.
\end{align*}
\]

(9)

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 \( \text{Poly1305}_R(D_0, \ldots, D_{m-1}) \), we have

\[
\text{Poly1305}_R(D_0, \ldots, D_{m-1}) = \text{Poly1305}_R(C_0, \ldots, C_{\ell-1}).
\]

(10)

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

\[
\begin{align*}
\text{Poly1305}_R(D_0, \ldots, D_{m-1}) &= R^4 \text{poly}_{R^4}(D_0, D_4, \ldots, D_{m-4}) \\
&+ R^3 \text{poly}_{R^3}(D_1, D_5, \ldots, D_{m-3}) \\
&+ R^2 \text{poly}_{R^2}(D_2, D_6, \ldots, D_{m-2}) \\
&+ R^1 \text{poly}_{R^1}(D_3, D_7, \ldots, D_{m-1})
\end{align*}
\]

(11)

In a manner similar to (6), define

\[
\begin{align*}
R &= (R^4, R^3, R^2, R)^T; \\
R_4 &= (R^4, R^3, R^1, R^4)^T; \\
D_i &= (D_{4i}, D_{4i+1}, D_{4i+2}, D_{4i+3})^T; \text{ for } i = 0, \ldots, m' - 1.
\end{align*}
\]

(12)
The computation in (11) is described in vector form in Algorithm 2.

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

<table>
<thead>
<tr>
<th>Input: ((D_0, \ldots, D_{t-1}))</th>
<th>Output: Poly1305(<em>R)((D_0, \ldots, D</em>{t-1}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (T \leftarrow D_0);</td>
<td></td>
</tr>
<tr>
<td>2 for (i = 1 ) to (m' - 1) do</td>
<td></td>
</tr>
<tr>
<td>3 (T \leftarrow R_4 \circ T + D_i)</td>
<td></td>
</tr>
<tr>
<td>4 (T \leftarrow R \circ T)</td>
<td></td>
</tr>
<tr>
<td>5 return (T_0 + T_1 + T_2 + T_3)</td>
<td></td>
</tr>
</tbody>
</table>

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 \(\text{vecMult}(\cdot, \cdot)\) followed by \(\text{vecReduce}(\cdot)\).

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 \(\text{vecMult}\) 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 \(R_4 \circ D_0\). Consider \(D_0 = (D_0, D_1, D_2, D_3)^T\). Depending on the value of \(\rho\), there are four cases.

\[
D_0 = \begin{cases} 
(C_0, C_1, C_2, C_3)^T & \text{if } \rho = 0; \\
(0, 0, 0, C_0)^T & \text{if } \rho = 1; \\
(0, 0, C_0, C_1)^T & \text{if } \rho = 2; \\
(0, C_0, C_1, C_2)^T & \text{if } \rho = 3. 
\end{cases}
\]  

(13)

Suppose \(\rho = 1\) so that \(D_0 = (0, 0, 0, C_0)\). Let the 5-limb representation of \(C_0\) be given by \((c_0, 4, \ldots, c_0, 0)\). Consider the schematic of the operation \(\text{vecMult}\) as discussed in Section 3.1. The representation of \(D_0\) in the 3 256-bit words \(t_0, t_1\) and \(t_2\) will look as follows (where \(x\) is a don’t care value).

\[
\begin{array}{cccccccc}
t_0 & c_{0,2} & c_{0,0} & 0 & 0 & 0 & 0 & 0 \\
t_1 & c_{0,3} & c_{0,1} & 0 & 0 & 0 & 0 & 0 \\
t_2 & x & c_{0,4} & x & 0 & x & 0 & x \\
\end{array}
\]

Since a lot of entries in the above representation are zeros, applying the Goll-Gueron \(\text{vecMult}\) operation to \(R_4\) and this \(D_0\) 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 \(D_0\) 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 $D_0 = (0, 0, C_0, C_1)$. When $\rho = 3$, $D_0 = (0, C_0, C_1, C_2, C_3)$, and in this case, the number of zeros is not sufficient to provide any improvement by using a multiplication algorithm different from $\text{vecMult}$. Below we provide the top level schematics of the improved initial multiplication algorithms for the cases of $\rho = 1$ and $\rho = 2$.

**Representation of $D_0$ for the case $\rho = 1$:** In this case, $D_0 = (0, 0, 0, C_0)$ is represented using 2 256-bit words as follows.

<table>
<thead>
<tr>
<th>$t_0$</th>
<th>$c_{0,3}$</th>
<th>0</th>
<th>$c_{0,2}$</th>
<th>0</th>
<th>$c_{0,1}$</th>
<th>0</th>
<th>$c_{0,0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The 5 256-bit words holding the output of $\text{vecMult}(R_4, D_0)$ in this case will have the following form.

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

**Representation of $D_0$ for the case $\rho = 2$:** In this case, $D_0 = (0, 0, C_0, C_1)$ is represented using 2 256-bit words as follows.

<table>
<thead>
<tr>
<th>$t_0$</th>
<th>$c_{0,3}$</th>
<th>$c_{1,3}$</th>
<th>$c_{0,2}$</th>
<th>$c_{1,2}$</th>
<th>$c_{0,1}$</th>
<th>$c_{1,1}$</th>
<th>$c_{0,0}$</th>
<th>$c_{1,0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The 5 256-bit words holding the output of $\text{vecMult}(R_4, D_0)$ in this case will have the following form.

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

In both the cases, the representation of $R_4$ using 2 256-bit words $r_0$ and $r_1$ 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 $S$ 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 $\text{vecReduce}$ algorithm mentioned in Section 3.1 is applied to $S$ to obtain the result $R_4 \circ D_0$. The output of $\text{vecReduce}$ is in the form of 3 256-bit words which is stored in $T$. The further multiplications $R_4 \circ T$ in the loop at Step 1 of Algorithm 2 are performed using the algorithm $\text{vecMult}$ and $\text{vecReduce}$.

**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 $\text{_mm256_mul_epu32}$ 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.
### Table 2: Operation counts for the modified multiplication algorithms for the cases \( \ell \mod 4 = 1 \) or 2.

<table>
<thead>
<tr>
<th>Name of Intrinsic</th>
<th>Count for ( \ell \mod 4 = 1 )</th>
<th>Count for ( \ell \mod 4 = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_mul_epi32</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>_mm256_set_epi32</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>_mm256_set_epi64x</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>_mm256_add_epi64</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>_mm256_permutevar8x32_epi32</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>_mm256_permute4x64_epi64</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>_mm256_unpacklo_epi64</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>_mm256_unpackhi_epi64</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>_mm256_blend_epi32</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>_mm256_shuffle_epi32</td>
<td>-</td>
<td>6</td>
</tr>
</tbody>
</table>

#### 4.2 Lazy Reduction

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

#### 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 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. The modified code is publicly available at the following link.

https://github.com/Sreyosi/Improved-SIMD-Poly1305

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. Performances of the the new code and that of the code accompanying [6] were measured using the same strategy.

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. The compile command used was the following:

```
gcc -mavx2 -O3 -fomit-frame-pointer
```

**Message length:** For measuring performance and comparison to [6], we considered messages with lengths up to 4KB\(^3\). 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. 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 
\[
su_l = 100\left(\frac{t_1 - t_0}{t_0}\right)
\]

The average speed-up is the average of all the \(su_l\)'s.

A top-level summary of the comparison is as follows.

- Haswell: speed-up has been obtained in 99.87% cases of 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.

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.

### Acknowledgement

We would like to thank Shay Gueron for kindly sharing the code associated with [6] with us and also for providing comments on an earlier version of this paper.

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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Range</th>
<th>Maximum Speed-up</th>
<th>Average Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell</td>
<td>49B - 1000B</td>
<td>29.70</td>
<td>12.06</td>
</tr>
<tr>
<td></td>
<td>1001B - 2000B</td>
<td>22.31</td>
<td>12.46</td>
</tr>
<tr>
<td></td>
<td>2001B - 4000B</td>
<td>15.15</td>
<td>9.06</td>
</tr>
<tr>
<td>Skylake</td>
<td>49B - 1000B</td>
<td>36.81</td>
<td>15.05</td>
</tr>
<tr>
<td></td>
<td>1001B - 2000B</td>
<td>15.24</td>
<td>6.94</td>
</tr>
<tr>
<td></td>
<td>2001B - 4000B</td>
<td>12.66</td>
<td>3.29</td>
</tr>
<tr>
<td>Kaby Lake</td>
<td>49B - 1000B</td>
<td>35.33</td>
<td>13.12</td>
</tr>
<tr>
<td></td>
<td>1001B - 2000B</td>
<td>21.49</td>
<td>12.94</td>
</tr>
<tr>
<td></td>
<td>2001B - 4000B</td>
<td>21.17</td>
<td>10.51</td>
</tr>
</tbody>
</table>

References


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.