Fast Software Implementation of Offset Merkle-Damgård (OMD)

Ralph Ankele, Robin Ankele

School of Computer and Communication Sciences
Summer@EPFL

August 21, 2014

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Dr. Reza Reyhanitabar
EPFL / LASEC
Abstract. Offset Merkle-Damgård (OMD) is an Authenticated Encryption scheme which is part of the ongoing CAESAR competition. This paper deals with the optimization of the current reference version of OMD. To optimize the reference version we analyzed the current version to identify bottlenecks in the implementation. In our analysis we show that the keyed compression function used in OMD takes the most computation time. We consider different optimized SHA2 implementations to overcome these limitations. Therefore, we use the Intel instruction sets for Streaming SIMD Extensions (SSE4) and the Advanced Vector Extensions (AVX1). We than compared our results of the optimized OMD versions with the reference OMD version and achieve an improvement of a factor 2 for large message sizes and a factor 2.5 for small message sizes.

Keywords: Offset Merkle-Damgård, CAESAR, optimization, SHA2, intrinsic, AVX, SSE4, Intel SHA extensions

1 Introduction

An Authenticated Encryption (AE) scheme provides confidentiality, integrity and authenticity simultaneously. These goals are achieved by combining separate cryptographic primitives, an encryption scheme to ensure confidentiality and a Message Authentication Code (MAC) for integrity and authenticity. However, the combination of a confidentiality mode with an authentication mode leads to implementation errors and is not efficient (e.g. the input stream has to be processed twice, for the encryption scheme and the MAC). Recently, some practical attacks have confirmed the vulnerability of these AE schemes [CHVV03] [Vau02].

To overcome these attacks and to achieve fast and efficient AE schemes, which should offer advantages over AES-GCM and should be suitable for widespread adoption, the CAESAR competition [cae] was announced. Our scheme, which is a CAESAR candidate, is called Offset Merkle-Damgård (OMD) [SDŞD+14] is a keyed compression function based mode of operation for nonce-based Authenticated Encryption with Associated Data (AEAD). We recommend two keyed compression functions from the SHA2 family, namely Sha-256 and Sha-512, where the parameterized compression functions will be called OMD-sha256 and OMD-sha512.

One of the most common ways of constructing an AE scheme is to define a mode of an existing block cipher (e.g. GCM [MV04], CCM [WHF03], OCB3 [TP11], COPA [ABL⁺13], OTR [Min13], McOE [FFLW11]). Other approaches are permutation based (e.g. SpongeWrap [BDPA11] and APE [ABB⁺13]) or hash function based (e.g. Hash-CFB [FLMW12]). Additionally, AE schemes can be separated between nonce-misuse resistant AE schemes and
nonce based AE schemes. First ones maintain authenticity and integrity up to a certain common prefix of the message, even if the nonce is repeated. Second ones lose all integrity and authenticity if the nonce is repeated. Some nonce-misuse resistant schemes are McOE and COPA, whereby nonce based schemes are GCM, OCB3, OTR, CCM and COBRA. The areas of application for Authenticated Encryption schemes is in SSH, IPSEC, SSL/TLS. Performance is an important factor as an AE scheme is usually used to process a high amount of data. In 2011, Intel therefore launched their Westmere micro architecture which implements dedicated instructions for AES [Gue12] that can be used for instance for AES-GCM. Another improvement should come in 2015 with the Skylake micro architecture, where Intel announces the Intel SHA extensions [GGY+13], with seven new-dedicated instructions for Sha-1 and Sha-256.

Current submissions to the CAESAR competition are mainly reference implementations whose are targeted for easy understanding of the underlying primitives and to simplify the scheme. In this paper, we present optimized versions of the CAESAR candidate OMD. Therefore, we take advantage of the new Intel CPU instruction sets and compiler optimization flags. More details about the optimization are discussed in Section 3.

**Related Work.** Recently, Andrey Bogdanov et. al. published a paper about AES-Based Authenticated Encryption Modes in Parallel High-Performance Software [BLT14]. In their paper they compare 12 AE schemes, where some of them are CAESAR candidates. In their work, they optimized the AE schemes for the new Intel Haswell micro architecture and used the AVX and AVX2 CPU instruction sets as well as improved AES-NI instructions. In 2011, Ted Krovetz and Phillip Rogaway published a paper about Software Performance of Authenticated-Encryption Modes [TP11]. In their paper they compared GCM, CCM, OCB1-3 and CTR where they optimized OCB for performance.

**Contribution and Outline.** This paper discusses several optimization techniques that are applied to the optimized versions of OMD. We recommend the usage of Sha-256 and Sha-512 as keyed compression functions. In the optimized OMD versions, highly efficient implementations of Sha-256 and Sha-512 are used. These are optimized for Intel processors with a micro architecture of Wolfdale and newer. The optimized versions use dedicated instructions from the CPU instruction sets of the Streaming SIMD Extensions 4 (SSE4) as well as the Advanced Vectors Extensions (AVX1) to increase the performance. Additionally, some compiler optimization techniques are used to improve the performance. In the end, we compared the results from the optimized OMD versions with each other and the reference version.

The remainder of this paper is organized as follows. Section 2 will give a brief description of the Offset Merkle-Damgård scheme. Furthermore, in Section 3 the Fast Software Implementation of OMD is discussed in more detail. Firstly, the main components are analyzed regarding their probability of occurrence and ordered by their runtime. Secondly, some optimization techniques are discussed, followed by some compiler optimization techniques. In the end of Section 3, some advanced optimization techniques are shown like intrinsic and the Intel SHA-Extensions are mentioned. Section 4 deals with our results, the measurement setup and some comparisons between different AEAD schemes. Finally, our results are discussed and an outlook to future work is given.
2 Short Description of Offset Merkle Damgård

In this section, we give some notations used throughout the paper, a short description of the recommended compression function of OMD and a description of the Offset Merkle-Damgård scheme.

Notations. We denote a set of binary strings with length of \( n \) bits (for some positive integer \( n \)) as \( \{0,1\}^n \). If the length is variable, but with finite length the set is denoted as \( \{0,1\}^* \). The concatenation of two strings \( X \) and \( Y \) is denoted as \( X||Y \). A substring of the string \( X = X_{m-1} \cdots X_0 \), is denoted as \( X[i \cdots j] \) for \( 0 \leq j \leq i \leq m - 1 \). Let \( |X| \) be the number of elements in a binary string \( X \), with \( \epsilon = |X| \) if \( X \) is a empty string. Furthermore, the notation of the operation \( \oplus \) is the bitwise XOR of the two strings \( X = X_{m-1} \cdots X_0, Y = Y_{m-1} \cdots Y_0 \). For any string \( X \) we have \( X \oplus \epsilon = \epsilon \oplus X = \epsilon = X \oplus X \).

OMD makes use of the multiplication of points in the Galois Field, which we denote as \( GF(2^n) \). OMD uses a keyed compression function which is defined as \( F : K \times \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n \), where \( n \) and \( m \) are two positive integers with \( n \) is the length of the hash digest and \( m \) is the length of the message input of the compression function. The key \( K \in \mathcal{K} \) of the keyed compression function is a non-empty set of strings, the digest of the compression function \( H \in \{0,1\}^n \) and the message \( M \in \{0,1\}^m \). The compression function can be seen as two arguments function \( F_K(H,M) = F(K;H,M) \). If the length of the key \( |K| = \epsilon \) we speak of a keyless compression function. We can build a keyed compression function from a keyless compression function by applying simple operations on the input parameters like bitwise XOR of the message and the key (i.e. \( M \oplus K \)) or by simply appending the message to the key (i.e. \( K||M \)). For example, for \( \text{sha256} \) we have \( \{0,1\}^{256} \times \{0,1\}^{512} \rightarrow \{0,1\}^{256} \), where we borrow \( k \) bits from the message input \( m \) to append the key \( K \).

OMD builds upon a compression function, and is a nonce-based authenticated encryption scheme for associated data (AEAD). This symmetric key scheme \( \pi = (\mathcal{K}, \mathcal{E}, \mathcal{D}) \) consists of a non-empty finite set \( \mathcal{K} \), an encryption algorithm \( \mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \rightarrow \mathcal{C} \cup \{\bot\} \) and a decryption algorithm \( \mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C} \rightarrow \mathcal{M} \cup \{\bot\} \), where \( K \in \mathcal{K} \) is the key, \( N \in \mathcal{N} \) is a nonce, \( A \in \mathcal{A} \) is the associated data, \( M \in \mathcal{M} \) is the message, \( C \in \mathcal{C} \) is the cipher text and \( \bot \) is a symbol to indicate an error. For correctness the following must hold for all \( M : \mathcal{D}_K(N,A,\mathcal{E}_K(N,A,M)) \). In an AEAD scheme the length of the message and associated data can be arbitrary long, but finite (i.e. \( |M| = \{0,1\}^*, |A| = \{0,1\}^* \)). The length of the nonce and the secret key are bounded to some fixed values (i.e. \( |K| = \{0,1\}^k, |N| = \{0,1\}^k \)). In OMD the cipher text \( C \) consists of \( C||\text{Tag} \), where \( C \) is the cipher text of the encryption with \( |C| = |M| \) and \( \text{Tag} \) is the tag of the authenticated encryption scheme with \( |\text{Tag}| = \tau \).

Furthermore, in OMD the function \( \text{ntz} \) denotes the number of trailing zeros of a string \( X \) (i.e. the number of rightmost bits that are zero).

2.1 The Compression Function

The authenticated encryption scheme OMD is a mode of operation for a keyed compression function. If the compression function is not keyed by itself it can be, by just concatenating the key to the message or simply by bitwise XOR of the key to the message. Compression functions are one of the most widely used and best-analyzed cryptographic primitives. Due to several years of research and the public NIST competition to find a new SHA-3 hash function, one can easily take advantage of a secure and reliable compression function.
Furthermore, Intel recently announced that they will support new instructions gaining performance acceleration for the SHA-1 and SHA-2 family.

Therefore, the designers of OMD recommend the two members of the SHA-2 family, sha-256, sha-512 as compression functions of OMD. Sha-256 can be used for 32-bit implementations and sha-512 for 64-bit implementations, respectively.

The SHA-2 family was published in 2001 by NIST as a secure hash function standard. The family consists of the following members (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256). It is a Merkle-Damgård construction and builds upon the SHA-1 family. The SHA-2 family is build upon the simple operations of bitwise xor, bitwise and, rotations, shifts and modular additions.

SHA-256. Operates over 64 rounds on eight 32-bit state words. It outputs a 256-bit digest and inputs a key $K \in \{0,1\}^{256}$ and a message $M \in \{0,1\}^{512}$.

SHA-512. Operates over 80 rounds on eight 64-bit state words. It outputs a 512-bit digest and inputs a key $K \in \{0,1\}^{512}$ and a message $M \in \{0,1\}^{1024}$.

The SHA-2 family is widely used in secure applications like SSH, SSL/TLS and IPSec.

2.2 Offset Merkle-Damgård

OMD, short for Offset Merkle-Damgård, is a nonce based authenticated encryption scheme with associated data, which was recently submitted by Reyhanitabar et al. [SDŞD+14] to the CAESAR competition to find a portfolio of authenticated encryption algorithms. The OMD family builds upon a keyed compression function. For the sake of convenience the designer of OMD recommend the current hash function standard SHA-2 for the use of the compression function of OMD. These two versions, namely omd_sha256, omd_sha512 can be used for 32-bit implementations and 64-bit implementations, respectively. OMD is provable secure in the standard model of the underlying well-established secure compression function. It has promising performance due to the recently announced Intel SHA Extensions and it builds upon simple operations like bitwise XOR, bitwise and, rotations, shifts and modular addition. Moreover, it gains nearly optimal performance in the number of compression function calls, that one can expect from an AEAD scheme that is only using a compression function. Furthermore, it has a high quantitative level of security due to the fact that it is based on a compression function with a large size hash size compared to AES based AEAD schemes which must rely on a block size of 128 bits.

OMD as an authenticated encryption scheme supports nonces as public message number and doesn’t support secret message numbers as proposed in the CAESAR call for submissions. The OMD mode of operation is as follows. One has to specify a keyed compression function $F_K : \{(0,1)^m \times \{0,1\}^n \} \rightarrow \{0,1\}^n$ and a tag length $\tau \leq n$. The message size has to be $m \leq n$, where the optimal choice is $m = n$. The encryption algorithm $OMD(F_K, \tau)$ of OMD takes four input parameters (secret key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^{\lceil N \rceil}$, associated data $A \in \{0,1\}^*$, message $M \in \{0,1\}^*$) and outputs the cipher text concatenated with the tag $\langle C||Tag, C \in \{0,1\}^{\lceil M \rceil} \rangle$ and $\langle Tag = \tau \rangle$. The decryption algorithm $OMD(F_K, \tau)$ of OMD behaves nearly the same, it takes four input parameters(secret key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^{\lceil N \rceil}$, associated data $A \in \{0,1\}^*$, cipher text concatenated with $\langle C||Tag \in \{0,1\}^{\lceil M \rceil+\tau} \rangle$ and outputs the message $M \in \{0,1\}^{\lceil C \rceil-\tau}$ or an error symbol $\bot$ indicating an error in decryption or verification of the tag.

The design of OMD is illustrated in Figure 1. Furthermore, the pseudo code of OMD is given in Listing 7 in the Appendix. As shown in the pseudocode OMD consists of four algorithms namely INITIALIZE, HASH, ENCRYPT and DECRYPT.
**Initialize** precomputes the \( L[i] \) values used for the message masking, where \( 0 \leq i \leq \log_2(l_{\text{max}}) \) and \( l_{\text{max}} \) is the maximum number of blocks that can be encrypted or decrypted.

**Hash** computes one half of the Tag over the associated data \( A \).

**Encrypt** computes the cipher text \( C \) and the second half of the Tag, which is then xored to the first half of the Tag, over the message \( M \).

**Decrypt** computes the message \( M \) from the cipher text \( C \| \text{Tag} \).

The masking values \( \triangle_{N,i,j} \) for the message and \( \bar{\triangle}_{i,j} \) for the associated data can be computed as follows.

\[
\begin{align*}
\triangle_{N,0,0} &= F_K(N||10^{n-1-|N|}, 0^n) \\
\text{for } i &\geq 1 \text{ do} \\
\triangle_{N,i,0} &= \triangle_{N,i-1,0} \oplus L[\text{ntz}(i)] \\
\triangle_{N,i,1} &= \triangle_{N,i,0} \oplus 2L^* \\
\triangle_{N,i,2} &= \triangle_{N,i,0} \oplus 3L^*
\end{align*}
\]

\[
\begin{align*}
\bar{\triangle}_{0,0} &= 0^n \\
\text{for } i &\geq 1 \text{ do} \\
\bar{\triangle}_{i,0} &= \bar{\triangle}_{i-1,0} \oplus L[\text{ntz}(i)] \\
\bar{\triangle}_{i,1} &= \bar{\triangle}_{i,0} \oplus L^*
\end{align*}
\]

In the following the settings for the OMD versions are described.

OMD\_sha256 uses \texttt{sha-256} as underlying compression function. The block length is \(|M_i| = 256\) bits. If needed, \( M_i \) is padded with \( 10^n \) to make it exactly 256-bits. The key is between \( 80 \leq k \leq 256 \) bits. If needed, \( K \) is padded with \( 0^{256-k} \). The nonce \( N \) can be between \( 96 \leq |N| \leq 255 \) bits. It is always padded with \( 10^{255-|N|} \). The associated data length is \(|A_i| = 512\) bits. If needed, \( A_i \) is padded with \( 10^n \) to make it exactly 512-bits. Finally, the Tag length is between \( 32 \leq \tau \leq 256 \) bits.

OMD\_sha512 uses \texttt{sha-512} as underlying compression function. The block length is \(|M_i| = 512\) bits. If needed, \( M_i \) is padded with \( 10^n \) to make it exactly 512-bits. The key is between \( 80 \leq k \leq 512 \) bits. If needed, \( K \) is padded with \( 0^{512-k} \). The nonce \( N \) can be between \( 96 \leq |N| \leq 511 \) bits. It is always padded with \( 10^{511-|N|} \). The associated data length is \(|A_i| = 1024\) bits. If needed, \( A_i \) is padded with \( 10^n \) to make it exactly 1024-bits. Finally, the Tag length is between \( 32 \leq \tau \leq 512 \) bits.

### 3 Fast Software Implementation of OMD

In this section, we discuss the techniques used for our fast software implementation of OMD. Furthermore, we give a detailed runtime analysis of the reference implementation and show how to improve this version.

**Motivation.** The CAESAR call for submission states the following:

* Each first-round submission must be accompanied by a portable reference software implementation to support public understanding of the cipher, cryptanalysis, verification of subsequent implementations, etc. [...] This implementation is expected to be optimized for clarity, not for performance; *

Currently, the submitted software implementations of the CAESAR competition are written for public understanding of the proposed cipher. Also the current OMD reference version is written in such a way. But for real life applications readability of code doesn’t matter, but performance matters. Therefore, it is important to optimize the current OMD version to a performance optimal version. In the following, we present different techniques how we optimized our OMD version.
Encrypting a message whose length is a multiple of the block length (i.e. $|M| = m$).
No padding is needed.

Encrypting a message whose length is not a multiple of the block length (i.e. $|M| < m$).
The final message block is padded to make it a full block.

Computing $\text{Tag}_a$ for an associate data whose length is a multiple of the input length (i.e. $|A_a| = n + m$).

Computing $\text{Tag}_a$ for an associate data whose length is not a multiple of the input length (i.e. $|A_a| < n + m$).
The final block is padded to make it a full block.

Fig. 1: Schematic of OMD. (Top) shows the encryption function where the input stream is a multiple of the OMD block size. (Middle) shows the encryption function where the input stream is padded to a multiple of the OMD block size. (Bottom Left) shows the hash function processing the associated data where the input stream is a multiple of the OMD block size. (Bottom Right) shows the hash function processing the associated data where the input stream is padded to a multiple of the OMD block size. (Right) shows the calculation of the Tag.

3.1 Analysis

Before one starts to optimize the current reference implementation, one must know where the weaknesses of the implementation are. Performance doesn’t improve much if we are targeting the wrong parts of the code. Therefore, we used the profiling tool Intel® VTune™ Amplifier XE 2013 to do a software performance analysis of the current reference implementation. The results of the performance analysis are given in Table 1. The performance analysis was conducted with our benchmarking method (details in Section 4.1). The resulting values are given in seconds (sec).

Regarding to Table 1 one can clearly see which functions should be optimized. The compression function (i.e. sha256, sha512) consume the most time, followed by the xor func-
Table 1: Performance Analysis of Reference Implementations [in seconds (sec)]

<table>
<thead>
<tr>
<th>Function</th>
<th>OMD-sha256</th>
<th>OMD-sha512</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall</td>
<td>73.797</td>
<td>55.504</td>
</tr>
<tr>
<td>f : compression</td>
<td>54.332</td>
<td>37.380</td>
</tr>
<tr>
<td>f : xor</td>
<td>12.526</td>
<td>11.442</td>
</tr>
<tr>
<td>f : multGF(2^n)</td>
<td>4.521</td>
<td>4.696</td>
</tr>
<tr>
<td>f : encrypt</td>
<td>0.675</td>
<td>0.549</td>
</tr>
<tr>
<td>f : ntz</td>
<td>0.514</td>
<td>0.320</td>
</tr>
<tr>
<td>f : calcL_i</td>
<td>0.385</td>
<td>0.184</td>
</tr>
<tr>
<td>f : maskingValues</td>
<td>0.173</td>
<td>0.170</td>
</tr>
<tr>
<td>f : hash</td>
<td>0.011</td>
<td>0.008</td>
</tr>
</tbody>
</table>

tion and the $GF(2^n)$ multiplication. These functions are repeatedly called in a for loop. Other functions like encrypt, hash or maskingValues are only called once and therefore they don’t consume that much time. Furthermore, it must be noticed that the compression function SHA as well as OMD itself is a serial cipher. This means that data in round $i$ depends on data in round $i-1$. Therefore, an optimized OMD implementation is not parallelizable. In the following, we present several techniques to improve the performance. Furthermore, we use optimized SHA-2 implementations to improve the performance of the reference implementation.

3.2 Optimization Techniques

There are several optimization techniques to optimize the reference implementation. In the following we are going to explain the C language optimization techniques we are using for our fast software implementation.

Precomputed values. To optimize for execution speed some values can be precomputed instead of calculating them at runtime. In case of OMD the values for $L[]$ and $L^*$ can be precomputed and stored in a static array. Due to the dependence of the key the values for $L[]$ and $L^*$ only have to be recomputed if the key changes. The reference implementation doesn’t implement the INITIALIZE algorithm, but computes the $L[]$ and $L^*$ values on the fly, during runtime. For the fast software implementation, we have optimized the INITIALIZE algorithm even further, to precompute exactly the needed $L[]$ values, and if in a further message more $L[]$ values are required, it just adds the missing values. Moreover, the $\log_2(l_{max})$ values are precomputed and stored in a static array. The optimized initialization is given in Listing 3.2.

```c
1: Algorithm INITIALIZE(K)
2: if K != K_int then
3:     K_int ← K, L_size ← 0
4:     /* calculations in GF(2^n)*/
5:    L_0 ← F_K(0^n, 0^n), L_1 ← 2L_0, L_2 ← 3L_0
6:    L[0] ← 4L_0
7: return
```

Furthermore, it is possible to precompute the masking values $\Delta_{N, i, j}$ for the message and $\bar{\Delta}_{i, j}$ for the associated data. The $\Delta_{i, j}$ values are only depending on the $L[]$ values that

```c
1: Algorithm PRECOMPUTE_L(l_start, l_max)
2: if l_max < L_size then
3:     return
4:     L_size ← l_max
5: for i ← l_start to log_2(l_max) do
6:     L[i] = 2L[i-1]  // doubling in GF(2^n)
7: return
```
are again only depending on key changes. Therefore, the $\tilde{\Delta}_{i,j}$ values only need to be recomputed if the key changes. The $\Delta_{N,i,j}$ values are depending on the nonce. Therefore only the first $\Delta_{N,0,0}$ has to be computed for every message. The other $\Delta_{N,i,j}$ with $i > 0, j > 0$ can be partly precomputed. As all of the $\Delta_{N,i,j}$ depends on the previous $\Delta_{N,i-1,j}$ one can precompute the $L[]$ and $L^*$ values and only xor the first $\Delta_{N,0,0}$ to the computed sum of $L[]$ values.

\[
\begin{align*}
\tilde{\Delta}_{0,0} &= 0^n \\
\tilde{\Delta}_{1,0} &= \tilde{\Delta}_{0,0} \oplus L[ntz(1)] = 0^n \oplus L[ntz(1)] \\
\tilde{\Delta}_{2,0} &= \tilde{\Delta}_{1,0} \oplus L[ntz(2)] = 0^n \oplus L[ntz(1)] \oplus L[ntz(2)] \\
\tilde{\Delta}_{3,0} &= \ldots \\
\Delta_{N,0,0} &= F_K(N||10^{n-1-|N|}, 0^m) \\
\Delta_{N,1,0} &= \Delta_{N,0,0} \oplus L[ntz(1)] = F_K(N||10^{n-1-|N|}, 0^m) \oplus L[ntz(1)] \\
\Delta_{N,2,0} &= \Delta_{N,1,0} \oplus L[ntz(2)] = F_K(N||10^{n-1-|N|}, 0^m) \oplus L[ntz(1)] \oplus L[ntz(2)] \\
\Delta_{N,3,0} &= \Delta_{N,2,0} \oplus L[ntz(3)] = F_K(N||10^{n-1-|N|}, 0^m) \oplus L[ntz(1)] \oplus L[ntz(2)] \oplus L[ntz(3)] \\
\Delta_{N,4,0} &= \ldots
\end{align*}
\]

Listed above is the precomputation of the first $\tilde{\Delta}_{i,j}, \Delta_{N,i,j}$ values. One can easily see that the nonce depending part can just be xorred to the precomputed $L[]$ value sum.

**SHA-Endianess.** Always, before the compression function (*i.e.* sha256, sha512) is called the chaining value is converted to big endian, then the compression function is invoked, and afterwards the digest is converted back to little endian. To circumvent these frequent conversions from little to big endian and vice versa, it is possible to bring the initial digest in big endian format and convert it back to little endian only at the output of the cipher (*i.e.* at the computation of the ciphertext $C_i \leftarrow M_i \oplus Digest_i$). Therefore, time for $n/2$ conversions between little to big endian and vice versa, can be saved.

**Loop unrolling.** Manual loop unrolling can be more efficient in some cases. Loop unrolling optimizes the execution speed at the expense of its binary size. With loop unrolling one analyzes the loop and interprets the iterations into a sequence of instructions which will reduce the loop overhead.

**Macros.** To save the overhead of function invocation some simple functions are replaced as C macros which are inserted from the preprocessor at compile time.

**Inline Functions.** Inline functions are substituted at the point where they were called and the inline function is directly inserted at the address of the function call, saving the overhead of a function invocation and return. This approach is very similar to the Macros, but in some cases inline functions serve better than a macro.

**Declarations, Defines and Datatypes.** To write an efficient and convenient implementation declarations, defines and datatypes should also be optimized. The optimized fast software implementation of OMD is rewritten with the same standard int and datatypes, so there is no need for conversions from or to other datatypes. Furthermore, instead of unnecessary memcpy, memset operations, all variables are declared with initial values. Moreover,

...
often it is faster if the datatypes/arrays consists of \(2^n\) elements. In our optimized version, all non-evaluated expressions are precomputed as far as possible, to save computation time.

**Pointers.** We could remove some unnecessary memory operations\((e.g.\, \text{memcpy}, \ \text{memset}, \ \text{memmove})\) from the reference implementation, by using pointers to the committed values.

**Inline Assembly.** To optimize the reference implementation even further, some specific code parts can be rewritten in inline assembly. Furthermore, the whole implementation can be optimized using intrinsics. More details about the usage of intrinsics are given in Section 3.4.

### 3.3 Compiler Optimizations

Performance optimization does not only depend on the written code, but also on the compiler used to translate the C code to machine code. There are several different compilers which all behave slightly different. Moreover, all compilers implement some optimization techniques themself, which can be used by the programmer.

For our measurements, we have used gcc-4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1). Additionally, we used the clang Apple LLVM version 5.1 (clang-503.0.40) compiler and the Intel® icc version 14.0.3 compiler. The results for those compilers are given in Table 2. The resulting values are in cycle per byte (cpb), where the measurement was performed on the avx1 optimized OMD version with \(|A| = 128\)-bit and \(|M| = 4096\)-bit. Every compiler supports different optimization levels. To activate those optimization levels it is necessary to pass the following flag to the compiler: `-O<number>`, where `number` is between 0..3. Furthermore, it is possible to select the `-Ofast` flag, which activates the optimization level O3 and additionally uses some not standard compliant math libraries. Details to the different optimization techniques behind the optimization levels can be found in [com].

<table>
<thead>
<tr>
<th>OMD version</th>
<th>gcc</th>
<th>clang</th>
<th>icc</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMD-sha256</td>
<td>27.15</td>
<td>28.29</td>
<td>26.91</td>
</tr>
<tr>
<td>OMD-sha512</td>
<td>21.32</td>
<td>22.88</td>
<td>18.37</td>
</tr>
</tbody>
</table>

### 3.4 Intrinsics

An intrinsic function is handled special by the compiler where the intrinsic function maps to some specific assembler instructions. Depending on the processor type and the used microarchitecture, different intrinsics are supported, mapping to different instruction set extensions. Intel provides for its \(x86\) architecture several extended vectorisation instruction sets to boost performance optimizations. In the following, we give a short overview about those instruction set extensions and show how we used them to improve the performance of the fast software implementation of OMD.
3.5 Extended Instruction Sets

The following extended instruction sets provide several new instructions for performance optimized implementations. For all instructions sets [int] gives a list of intrinsics and the associated assembly instructions.

SSE. The SSE extended instruction sets, short for Streaming SIMD Extensions, contain about 70 new instructions for high performance optimized implementations. SSE was subsequently expanded by SSE2, SSE3 and SSE4 providing new floating point math operations. The latest SSE versions are available for Intel processors with a micro architecture from Wolfdale and newer.

SSE originally provides eight 128-bit registers, namely xmm0-7, but supports for Intel 64 architecture eight more xmm registers, namely xmm8-15. Since SSE only supported a single datatype (i.e. four 32-bit floats) on the xmm registers, SSE2 expanded it even further (i.e. two 64-bit doubles, one 128-bit -), two 64-bit -, four 32-bit -, eight 16-bit -, sixteen 8-bit integers). The SSE extension supports several different instructions for all different purposes. [int] gives a list with all possible intrinsics that maps to the given SSE instructions.

The big advantage of the SSE instructions is the vectorization versus the scalar operations used in normal instructions. In a scalar mode it is only possible to execute instructions sequentially, one after each other. In vector mode, it is possible to execute one operation on a vector of data meaning to parallelizing the operation on multiple data. Therefore, when working on 32-bit registers, one can get a speedup up to 4 using vectorization.

![Fig. 2: Vectorizations versus Scalar. On the left is a operation in SIMD mode, on the right in Scalar mode.](image)

In our optimized OMD implementation we take advantage of the use of an Intel assembler based SHA implementation with SSE4 instructions. Figure 3 gives a comparison between different Sha-256 implementations, compiled with -Ofast. Using the SSE4 instructions we reach a speedup of 2 compared to the reference implementation. Compared to the non-optimized reference implementation we get a speedup of 3.8. For Sha-512 the comparison is given in Figure 4. It give a speedup of 1.7 for the -Ofast compiled reference implementation and a speedup of 3 for the non-optimized reference implementation.

Since the compression function sha-256, sha-512 is a serial cipher it is not parallelizable. But the message scheduling of sha can be parallelized using vectorization. For sha-256 four message words and for sha-512 two message words at a time can be processed. Since SSE and AVX does not implement vector shifts there must be done some workaround
to perform shift or rotation operations. The round function is executed in the integer execution unit in the processor. More information about the fast SHA implementation can be found in [GKG12], [GCG12]. Furthermore, we have implemented a version of OMD using the following SSE intrinsics: \_mm\_setzero\_si128(\ldots), \_mm\_set\_epi8(\ldots), \_mm\_loadu\_si128(\ldots), \_mm\_xor\_si128(\ldots), \_mm\_storeu\_si128(\ldots).

**AVX.** The AVX extended instruction sets, short for Advanced Vector Extensions, build upon the SSE instructions sets and expand those with new instructions and a new coding scheme. AVX1 operates on 256-bit registers and AVX2 expands also most integer commands to 256-bit. Furthermore, AVX512 introduces new 512-bit registers. The extended instruction sets of AVX1 are available for Intel processors with a micro architecture from Sandy Bridge. AVX2 is supported from Haswell and AVX512 is supported from Kings Landing and newer. The Kings Landing micro architecture will be available in processors beginning in 2015.

AVX extends the previous xmm0-15 128-bit registers to sixteen 256-bit registers, namely ymm0-15. Furthermore, it supports datatypes of eight 32-bit floats or four 64-bit doubles. AVX2 adds integer datatypes and AVX512 expands those to 512-bit. Moreover, memory alignments have been relaxed and three-operand non-destructive operations have been added. Previous, with two-operand commands the source operand was always overwritten by the instruction (e.g. \(A = A + B\)). With the new three-operand instructions (e.g. \(A = B + C\)) overheaded mov instructions to secure the source-value are not needed.

We have also used a fast SHA implementation based on AVX1 instructions. According to Figure 3 we get a speedup of 2.5 for a -Ofast compiled reference implementation for Sha-256 and a speedup of 5 for a non-optimized reference implementation. For Sha-512 shown in Figure 4 we get a speedup of 2.2 for the optimized and a speedup of 4.2 for the non-optimized version.

For the AVX based SHA implementation the same behaviour as for SSE4 applies. Some improvements using AVX instructions can be made to slightly improve this version over the SSE4 version. Additionally, we have implemented a version of OMD using the following AVX1 intrinsics: \_mm256\_extractf128\_si256(\ldots), \_mm\_setzero\_si128(\ldots), \_mm\_extract\_epi8(\ldots), \_mm256\_setr\_epi8(\ldots), \_mm\_or\_si128(\ldots), \_mm\_slli\_si128(\ldots), \_mm256\_setr\_m128i(\ldots), \_mm256\_setzero\_si256(\ldots), \_mm256\_loadu\_si256(\ldots), \_mm256\_castps\_si256(\ldots), \_mm256\_xor\_ps(\ldots), \_mm256\_castsi256\_ps(\ldots), \_mm256\_storeu\_si256(\ldots).

**AVX2.** As mentioned above AVX2 extends AVX by supporting integer datatypes for the 256-bit registers. It further adds vector shifts on 256-bit ymm register as new instructions. Due to the lack of a Haswell processor architecture we could not include the Intel provided fast SHA implementation using AVX2 instructions in our measurements. But we used the Intel® Software Development Emulator [emu] to verify our implementation. Using the AVX2 instructions it is possible to do vector shifts. Intel provides for sha-256 two different versions using the rorx instructions for vector shifts. The first one, namely avx2\_rorx2 is optimized for small buffer sizes and a small memory footprint and the second one, namely avx2\_rorx8 is optimized for large buffer sizes but a bigger memory footprint. Using these instructions should further improve the speed of the compression functions(sha-256, sha-512).

**AVX512.** AVX512 extends the previous AVX instructions sets by 32 512-bit registers zmm0-31. Furthermore, it supports 32-bit and 64-bit integers, floats and doubles. The
AVX512 instruction set can be used to further improve the speed of the compression functions (sha-256, sha-512).

Fig. 3: Sha-256 Implementations. Note: REF (black) is the reference version, OPENSSL (red) is the current openssl implementation, SSE4 (blue) uses Intel SSE4 cpu instructions, AVX1 (green) uses Intel AVX1 cpu instructions.

Fig. 4: Sha-512 Implementations. Note: REF (black) is the reference version, OPENSSL (red) is the current openssl implementation, SSE4 (blue) uses Intel SSE4 cpu instructions, AVX1 (green) uses Intel AVX1 cpu instructions.
3.6 SHA-Extensions

Intel recently announced in July 2013 to implement new SHA Extensions [GGY+13], supporting performance optimized instructions for the SHA-1 and 2 family (i.e. SHA-1, SHA-256). The Intel SHA extensions will be available in Intel processors with a micro architecture from Skylake and newer. These micro architecture will be available in processors beginning in mid. 2015.

The Intel SHA-Extensions offer 3 new 128-bit SSE based instructions for SHA256 (i.e. sha256msg1, sha256msg2 and sha256rnds2). The sha256msg1 and sha256msg2 instruction aims for the message schedule where sha256msg1 calculates the $\sigma_0(W_t-15) + W_t-16$ part of the message schedule. The sha256msg2 instruction calculates $\sigma_1(W_t-2) + W_t-7 + \sigma_0(W_t-15)+W_t-16$ to finale the message schedule. The round instruction, sha256rnds2, calculates two Sha-256 rounds at once. Additionally, the 32 bit working variables for SHA (C, D, G, H) are stored in one 128-bit XMM register and (A, B, E, F) are stored in another XMM register for simple assignment. The reduction to those instructions will give an enormous performance increase for a Sha-256 implementation.

Additionally, we have implemented a Sha-256 version using Intel SHA-Extensions. Because there is no processor available that supports Intel SHA-Extensions at the time of development, we have used the Intel Software Development Emulator (SDE) [emu] to get an estimation of the runtime. The results are given in Figure 5, where the REF, SSE4, AVX1 implementations are based on real measurements and the SHA-EXT implementation is an estimation using the Intel SDE. The estimations are based on the instructions count $\times$ latency for each instructions used in the execution of Sha-256. The latency values are based on a Haswell processor [Fog14]. The estimated performance for a message of $|M| = 4096$ bytes is around 4 cycles per byte. Therefore, we get a speedup compared to the best current Sha-256 implementation (i.e. AVX1) of 2.5. For an implementation of OMD-Sha-256 using Intel SHA-Extensions we estimate a runtime between 12-15 cycles per byte. For such an implementation the speedup compared to the non-optimized OMD-Sha-256 implementation is around 11.5 and 4.4 to a -Ofast optimized reference implementation and even 2.2 compared to the currently best implementation (i.e. using AVX1 extensions).

4 Results

In this section, we present the results of the optimized versions of OMD and compare them between each other.

4.1 Measurement Setup

In our benchmarking, we consider message sizes and associated data sizes from 0 to 4096 byte. In our 3D surface plots we visualize the different optimized OMD versions for a message length and associated data length from 128 to 2048 bytes. For the 2D plots we consider a message length from 0 to 4096 bytes and a fixed length of 0 bytes for the associated data. All measurements were taken on a 64-bit 2.4GHz dual-core Intel Core i5-2415M processor [pro]. The measurement was performed on a MacBook Pro Late 2011 with Ubuntu 12.10. As C compiler the gcc-4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1) compiler was used. The corresponding assembler was the GNU assembler version 2.22.90.20120924 (GNU Binutils for Ubuntu). As described in Section 3 additionally the compiler optimization flag -Ofast was used to enable compiler optimization. The measurement uses the
time stamp counter with the RDTSC instruction. The performance is measured in cycles per byte (cpb).
To get rid of the noise (e.g. context switches) during measurement and to get a clean and stable result, we use the same method as described in the paper from Krovetz and Rogaway [TP11] in their benchmarking. In their approach they determine the performance as the median of 91 averaged timings of 200 measurements each.

![Sha-256 Estimation](image)

**Fig. 5: Sha-256 Estimation.** Note: REF (black) is the reference version, SSE4 (blue) uses Intel SSE4 cpu instructions, AVX1 (green) uses Intel AVX1 cpu instructions. SHA-EXT (red) uses Intel Sha-Extensions cpu instructions, values are estimated.

### Table 3: omdsha256k128n96tau64v1

<table>
<thead>
<tr>
<th>Mode</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMD-ref</td>
<td>102.06</td>
<td>76.30</td>
<td>63.61</td>
<td>57.12</td>
<td>54.42</td>
<td>52.62</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>47.31</td>
<td>38.32</td>
<td>34.86</td>
<td>33.16</td>
<td>32.29</td>
<td>31.76</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>39.72</td>
<td>33.12</td>
<td>30.19</td>
<td>28.81</td>
<td>27.92</td>
<td>27.42</td>
</tr>
</tbody>
</table>

### Table 4: omdsha512k512n256tau256v1

<table>
<thead>
<tr>
<th>Mode</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMD-ref</td>
<td>109.62</td>
<td>73.11</td>
<td>54.40</td>
<td>45.64</td>
<td>40.88</td>
<td>38.64</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>47.40</td>
<td>33.75</td>
<td>28.37</td>
<td>25.62</td>
<td>24.36</td>
<td>23.64</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>41.82</td>
<td>29.34</td>
<td>24.58</td>
<td>22.44</td>
<td>21.39</td>
<td>20.76</td>
</tr>
</tbody>
</table>
Fig. 6: Comparison between optimized OMD-sha256 versions with increasing message length and associated data = 0. Note: REF (black) is the reference version, SSE4 (red) uses Intel SSE4 CPU instructions, AVX1 (blue) uses Intel AVX1 CPU instructions.

Fig. 7: Comparison between optimized OMD-sha512 versions with increasing message length and associated data = 0. Note: REF (black) is the reference version, SSE4 (red) uses Intel SSE4 CPU instructions, AVX1 (blue) uses Intel AVX1 CPU instructions.
Table 5: Performance Analysis of OMD-sha256 Implementations [in seconds (sec)]

<table>
<thead>
<tr>
<th>Function</th>
<th>OMD-ref</th>
<th>OMD-sse4</th>
<th>OMD-avx1</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall</td>
<td>73.797</td>
<td>17.538</td>
<td>15.308</td>
</tr>
<tr>
<td>$f :$ compression</td>
<td>54.332</td>
<td>14.156</td>
<td>11.534</td>
</tr>
<tr>
<td>$f :$ xor</td>
<td>12.526</td>
<td>1.864</td>
<td>2.280</td>
</tr>
<tr>
<td>$f :$ multGF($2^n$)</td>
<td>4.521</td>
<td>*1</td>
<td>*1</td>
</tr>
</tbody>
</table>

Table 6: Performance Analysis of OMD-sha512 Implementations [in seconds (sec)]

<table>
<thead>
<tr>
<th>Function</th>
<th>OMD-ref</th>
<th>OMD-sse4</th>
<th>OMD-avx1</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall</td>
<td>55.504</td>
<td>13.371</td>
<td>11.876</td>
</tr>
<tr>
<td>$f :$ compression</td>
<td>37.380</td>
<td>9.766</td>
<td>8.036</td>
</tr>
<tr>
<td>$f :$ xor</td>
<td>11.442</td>
<td>1.980</td>
<td>2.105</td>
</tr>
<tr>
<td>$f :$ multGF($2^n$)</td>
<td>4.696</td>
<td>*1</td>
<td>*1</td>
</tr>
</tbody>
</table>

4.2 Discussion

The best results for OMD-sha256 and OMD-sha512 are listed in Table 3 and in Table 4, respectively. The first column in the table specifies the AE mode; the other columns illustrate the performance in cycles per byte (cpb) for specific message lengths and associated data length = 0 bytes. The results for all specific versions of the OMD families can be found in the Appendix in tables 8 to 14.

As it can be seen in Table 3 the OMD-sha256 version optimized for SSE4 is about twice as fast as the reference version. For small messages sizes it is even faster where it’s about 2.5 times faster than the reference version. The OMD version optimized for AVX1 improves the performance even more. Table 4 visualizes the best results for the OMD-sha512 versions. The performance of OMD-sha512 is slower for small message sizes, but increases for large messages sizes when comparing OMD-sha256 with OMD-sha512. Again, the optimized versions for SSE4 and AVX1 are about 2 to 2.5 times faster than the reference OMD-sha512 version. On 64-bit machines, the performance of OMD-sha512 is higher than for OMD-sha256, because the Sha-512 has less rounds per byte than the Sha-256 when using 64-bit registers. More details can be found at [SG10].

Figure 6 and Figure 7 visualize the performance of the best results for OMD-sha256 and OMD-sha512. In the Appendix, there are additionally 3D surface plots for OMD-sha256 in Figure 8 for the reference version, Figure 9 for the optimized SSE4 version and Figure 10 for the optimized AVX1 version. There are also 3D surface plots for OMD-sha512 in Figure 11 for the reference version, Figure 12 for the optimized SSE4 version and Figure 13 for the optimized AVX1 version.

The 3D surface plots visualize the performance for an increasing message length and associated data length from 128 bytes to 2048 bytes. Furthermore, in Tables 5, 6 the performance analysis of the non-optimized OMD-sha256/sha512 reference version is compared to the optimized sse4/avx1 versions of OMD. The measurement method is again as mentioned in Section 4.1. The resulting values are in seconds. Overall, there is a huge performance improvement using the optimized OMD versions compared to the non-optimized reference versions.

\[1\] Values are too small to capture for Intel® VTune™ Amplifier XE 2013
implementation. For OMD-sha256 we get a speedup of 4.2 for sse4 and 4.8 for avx1 and for OMD-sha512 a speedup of 4.1 for sse4 and 4.7 for avx1.

5 Conclusions

In this paper, we have discussed the performance of the optimized versions of the authenticated encryption scheme Offset Merkle-Damgård (OMD), which is part of the ongoing CAESAR competition.

Firstly, we analyzed the current reference version of OMD to identify potential bottlenecks regarding performance. OMD builds upon a keyed compression function, where the designers recommend the SHA-2 family, special SHA-256 and SHA-512. The reference version was designed for clarity and easy understanding and therefore did not use many optimization techniques. We showed that the exchange of the current SHA implementation with an optimized implementation with the Intel Fast SHA versions for SSE4 and AVX1 brings a dramatic improvement of the performance. Additionally, we used compiler optimization flags like -Ofast and different C-compiler to improve the performance. Another improvement we showed is the pre-calculation of values so that the must not be computed during runtime. A main improvement for the performance could be established by using intrinsics, which optimized the performance on Intel processors.

In our results we compared the different optimized OMD versions and compared them with each other. Therefore, we have shown that the optimized SSE4 and AVX1 versions increased the performance with a factor 2 to 2.5. Additionally, we have shown that OMD-sha512 is faster than OMD-sha256 for large message sizes.

In 2013 Intel announces the SHA Extensions, which should be available from mid 2015 in the new Skylake micro architectures. The usage of these specific instructions will increase the performance of OMD by a large factor.

References


### Appendix

**Listing 7: Pseudo code of Offset Merkle-Damgård**

1: Algorithm Initialize(K)  
2: \(L^* \leftarrow F_K(0^n, 0^m)\)  
3: \(L[0] \leftarrow 4 \cdot L^* \text{ doubling in } GF(2^n)\)  
4: for \(i \leftarrow 1 \text{ to } [\log_2(l_{max})]\) do  
5: \(L[i] = 2 \cdot L[i-1] \text{ doubling in } GF(2^n)\)  
6: return

1: Algorithm HashK(A)  
2: \(b \leftarrow n + m\)  
3: \(A_1 || A_2 \cdots A_{l-1} || A_l \leftarrow A, \text{ where } |A_i| = b \text{ for } 1 \leq i \leq l - 1 \text{ and } |A_l| \leq b\)  
4: Tag_a \leftarrow 0^n  
5: \(\Delta \leftarrow 0^n\)  
6: for \(i \leftarrow 1 \text{ to } l - 1\) do  
7: \(\Delta \leftarrow \Delta + [l \text{ntz}(i)]\)  
8: Left \(\leftarrow A_i[b - 1 \cdots m]\); Right \(\leftarrow A_i[m - 1 \cdots 0]\)  
9: Tag_a \leftarrow Tag_a \oplus F_K(Left \oplus \Delta, \text{ Right})  
10: if \(|A_i| = b\) then  
11: \(\Delta \leftarrow \Delta + L[l \text{ntz}(i)]\)  
12: Left \(\leftarrow A_i[10^n-|A_i|\cdots b - 1 \cdots m]\); Right \(\leftarrow A_i[m - 1 \cdots 0]\)  
13: Tag_a \leftarrow Tag_a \oplus F_K(Left \oplus \Delta, \text{ Right})  
14: else  
15: \(\Delta \leftarrow \Delta + L\)  
16: Left \(\leftarrow A_i[10^n-|A_i|\cdots b - 1 \cdots m]\); Right \(\leftarrow A_i[m - 1 \cdots 0]\)  
17: Tag_a \leftarrow Tag_a \oplus F_K(Left \oplus \Delta, \text{ Right})  
18: return Tag_a

1: Algorithm EK(N, A, M)  
2: if \(|N| > n + l\) then  
3: return ⊥  
4: \(M_1 || M_2 \cdots M_{l-1} || M_l \leftarrow M, \text{ where } |M_i| = m \text{ for } 1 \leq i \leq l - 1 \text{ and } |M_l| \leq m\)  
5: \(\Delta \leftarrow F_K(N)[10^n-|N|0^m] \text{ initialize } \Delta_{N,0,0}\)  
6: \(H \leftarrow 0^n\)  
7: \(\Delta \leftarrow \Delta + L[0] \text{ compute } \Delta_{N,1,0}\)  
8: \(H \leftarrow F_K(H \oplus \Delta, (\tau)_m)\)  
9: for \(i \leftarrow 1 \text{ to } l - 1\) do  
10: \(M_i \leftarrow H \oplus C_i\)  
11: \(\Delta \leftarrow \Delta + [l \text{ntz}(i)]\)  
12: \(H \leftarrow F_K(H \oplus \Delta, M_i)\)  
13: \(M_i \leftarrow H \oplus C_i\)  
14: if \(|C_i| = m\) then  
15: \(\Delta \leftarrow \Delta + 2L^*\)  
16: Tag_a \leftarrow F_K(H \oplus \Delta, M_l)  
17: else if \(|C_i| \neq 0\) then  
18: \(\Delta \leftarrow \Delta + 3L^*\)  
19: Tag_a \leftarrow F_K(H \oplus \Delta, M_l)[10^n-|M_l|\cdots -1]  
20: return Tag_a
<table>
<thead>
<tr>
<th>Table 8: omdsha256k128n96tau96v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>102.70</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>49.71</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>42.83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 9: omdsha256k128n96tau128v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>102.77</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>49.72</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>42.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10: omdsha256k192n104tau128v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>105.00</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>57.20</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>43.53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 11: omdsha256k256n104tau160v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>102.63</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>49.69</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>44.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12: omdsha256k256n248tau256v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>103.29</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>50.64</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>43.84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 13: omdsha512k128n128tau128v1</th>
<th>Message length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>128</td>
</tr>
<tr>
<td>OMD-ref</td>
<td>109.57</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>48.05</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>42.51</td>
</tr>
</tbody>
</table>
Table 14: omdsha512k256n256tau256v1

<table>
<thead>
<tr>
<th>Mode</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMD-ref</td>
<td>111.01</td>
<td>73.92</td>
<td>55.43</td>
<td>46.10</td>
<td>41.65</td>
<td>39.14</td>
</tr>
<tr>
<td>OMD-sse4</td>
<td>51.17</td>
<td>36.48</td>
<td>31.04</td>
<td>28.15</td>
<td>26.96</td>
<td>26.16</td>
</tr>
<tr>
<td>OMD-avx1</td>
<td>42.03</td>
<td>29.88</td>
<td>24.99</td>
<td>22.64</td>
<td>21.59</td>
<td>20.83</td>
</tr>
</tbody>
</table>

Fig. 8: 3D surface plot of OMD-sha256 reference version with increasing message and associated data length.

Fig. 9: 3D surface plot of the OMD-sha256 version optimized for SSE4 with increasing message and associated data length.
Fig. 10: 3D surface plot of the OMD-sha256 version optimized for AVX1 with increasing message and associated data length.

Fig. 11: 3D surface plot of OMD-sha512 reference version with increasing message and associated data length.
Fig. 12: 3D surface plot of the OMD-sha512 version optimized for SSE4 with increasing message and associated data length.

Fig. 13: 3D surface plot of the OMD-sha512 version optimized for AVX1 with increasing message and associated data length.
6 How to run the optimized versions

6.1 Download

The source code of the optimized OMD versions is public available under TODO for OMD-sha256 and under TODO for OMD-sha512.

6.2 Dependencies

– The current implementation only supports Linux
– Gcc version 4.7.x or newer
– GNU assembler version 2.22.90.20120924 or newer
– CMake 2.8.1 or newer
– Matlab 2012b

6.3 How to build the optimized versions

There is a cmake support included in the project. Therefore it is quite easy to build and run the OMD framework.

First it is necessary to create a building directory:

```bash
mkdir _build
cd _build
```

After that you will need to run cmake and you also can specify some parameters to select which OMD/SHA version it will build:

```bash
cmake -D OMD_VERSION=<OMD_VERSION> -D SHA_VERSION=<SHA_VERSION> ..
```

See Table 15a and Table 15b for more details.

<table>
<thead>
<tr>
<th>OMD_VERSION</th>
<th>SHA_VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>omdsha256k128n96tau64v1</td>
<td>ref</td>
</tr>
<tr>
<td>omdsha256k128n96tau96v1</td>
<td>sse4</td>
</tr>
<tr>
<td>omdsha256k128n96tau128v1</td>
<td>avx1</td>
</tr>
<tr>
<td>omdsha512k128n128tau128v1</td>
<td>(b) SHA_VERSION</td>
</tr>
<tr>
<td>omdsha512k512n256tau256v1</td>
<td></td>
</tr>
</tbody>
</table>

If OMD_VERSION and/or SHA_VERSION are left empty it will just take the default version (first option from the tables). When cmake is finished you can build your targets.
Currently the following targets are supported:

```make
timing .......... Builds the target for time measurement
testing .......... Builds the target for testing
```

The corresponding executable are stored in the directory `/out/bin/`.

### 6.4 Example build and run

```bash
mkdir _build
cd _build
cmake -D OMD_VERSION=omdsha256k128n96tau64v1 -D SHA_VERSION=ref ..
make timing
./out/bin/timing
```

This example creates the `_build` folder and builds the framework for the OMD version: `omdsha256k128n96tau64v1` and the `ref` SHA version. As target the timing measurement is chosen (i.e. `make timing`). The timing measurement is started by running `./out/bin/timing`.

### 6.5 Visualize results using Matlab

To visualize the results you can use the matlab files provided in the `matlab` folder. Firstly, to visualize the 2D plots you have to use the `visualisation2D.m` file. Run the matlab file like:

```matlab
visualisation2D('PATH_TO_LOGFILE', NUMBER_OF_ENTRIES)
```

where you have to replace `PATH_TO_LOGFILE` with the path to the logfile returned when you run the timing (i.e. the logfile should be in the _build folder) and `NUMBER_OF_ENTRIES` with the number of entries in the logfile (e.g. if you run the timing for REF, SSE4 and AVX1 the number is 3).

Secondly, to visualize the 3D plots you have to use the `visualisation3D.m` file. Run the matlab file like:

```matlab
visualisation3D('PATH_TO_LOGFILE', NUMBER_OF_ENTRIES)
```

where you have to replace `PATH_TO_LOGFILE` with the path to the logfile returned when you run the timing (i.e. the logfile should be in the _build folder) and `NUMBER_OF_ENTRIES` with the number of entries in the logfile (e.g. if you run the timing for REF, SSE4 and AVX1 the number is 3).

### 6.6 Example visualization

```matlab
visualisation2D('log_omdsha256k128n96tau64v1', 3)
```

This example creates an 2D plot of the data for the OMD version: `omdsha256k128n96tau64v1`. The log file contains 3 entries, therefore the plot will visualize 3 curves (i.e. REF, SSE4, AVX1).