Speeding Up OMD Instantiations in Hardware

Diana Maimuţ$^{1[0000-0002-9541-5705]}$ and Alexandru Ştefan Mega$^{1,2[0000-0002-9541-1114]}$

^{1} Advanced Technologies Institute
10 Dinu Vintilă, Bucharest, Romania
`{diana.maimut,ati}@dcti.ro`
^{2} Politehnica University of Bucharest
Bucharest, Romania
`megastefanalexandru@gmail.com`

Abstract. Particular instantiations of the Offset Merkle Damgård authenticated encryption scheme (OMD) represent highly secure alternatives for AES-GCM. It is already a fact that OMD can be efficiently implemented in software. Given this, in our paper we focus on speeding-up OMD in hardware, more precisely on FPGA platforms. Thus, we propose a new OMD instantiation based on the compression function of BLAKE2b. Moreover, to the best of our knowledge, we present the first FPGA implementation results for the SHA-512 instantiation of OMD as well as the first architecture of an online authenticated encryption system based on OMD.

Keywords: Authenticated encryption, pseudorandom function, compression function, provable security, FPGA, hardware optimization, nonce respecting adversaries.

1 Introduction

Authenticated encryption (AE) primitives ensure both message confidentiality and authenticity. Initially, AE algorithms achieved confidentiality and integrity by combining two distinct cryptographic primitives (one for each of the two goals). Around two decades ago the perspective of having a unique primitive for confidentiality and integrity started to appear. Rogaway [16] extended AE schemes by adding a new type of input for associated data (AD) and, thus, AEAD (authenticated encryption with associated data) was the next step. Such a model is helpful in real world scenario in which part of the message (*e.g.* a header) needs only to be authenticated. We do not recall the technical aspects of AEAD schemes as it is outside the scope of our paper. We refer the reader to [16, 17] for a detailed description regarding the previously mentioned topic.

The *Competition for Authenticated Encryption: Security, Applicability, and Robustness* (CAESAR) started in early 2014 and finished in 2018 [1]. The *Offset Merkle-Damgård* (OMD) authenticated encryption scheme [2] was one of the CAESAR submissions. OMD is, in fact, an authenticated encryption mode of operation for keyed compression functions. The OMD instantiations presented in the original paper are based on the compression functions of two hash functions which are part of the SHA-2 family: SHA-256 and SHA-512.

OMD was accepted as a valid CAESAR submission for CAESAR and, thus, a process of public analysis from the community naturally followed. Given its characteristics which proved to be in accordance to the CAESAR requirements especially from the security point of view, OMD was further accepted as a second round candidate[3].

As stated in [2, 7], in the case of the original scheme's software implementations the speed can be considerably increased due to, *e.g.*, the performance acceleration instructions of INTEL's architecture processors. Thus, the implementation efficiency of the two OMD original instantiations becomes comparable with the AES-GCM one in software.

Given that from the security point of view OMD has more secure versions than AES-GCM and its software implementations are highly efficient, we believe that due to the lack of a competitive hardware implementation OMD did not make it until the third CAESAR round.

Therefore, especially in view of the diversity of secure authenticated encryption schemes we have to focus on providing practical implementations for them.

We pay particular attention to using the compression functions of SHA-512 and BLAKE2b both for a higher security level and a more hardware friendly word dimension.

Prior Work. A rather unoptimized hardware implementation of the original OMD scheme was submitted to CAESAR. Thus, considering the initial metrics, OMD seemed quite unattractive as compared to AES-GCM. Later on, in 2017, the authors of [8] presented their results regarding selected hardware implementations of CAESAR round 2 candidates. We aim at improving the previous results, providing the reader with better hardware implementation metrics of various OMD instantiations. The OMD implementation discussed in [8] is the primary recommendation submitted to CAESAR, *i.e.* using the compression function of SHA-256. Thus, to the best of our knowledge, we present the first FPGA implementation results for the SHA-512 instantiation of OMD.

Motivation. Our motivation for choosing OMD among all the CAESAR submissions is at least threefold. ① OMD's design is an exotic one. The scheme represents the only CAESAR proposal based on a compression function. ② When it comes to real world cryptographic applications and systems there may perfectly be some use cases in which security requirements are way higher than usual and the physical resources of an implementation platform are abundant. Nevertheless, our results become meaningful either in the case of an enhanced security context (*e.g.* secure government applications). ③ In [8] it is reported

[3] All the withdrawn schemes are listed on the competition's website. Almost all the withdrawn submissions are due to attacks reported by the community. It can easily be observed that for OMD no attack was presented.

that “OMD ends up near the bottom of Tp/A[4] ratios for all CAESAR Round Two candidates”. The explanation pointed out by the authors focuses on the big number of rounds of SHA-256 and, thus, the limited throughput. Our results show that we can obtain superior implementation metrics, especially considering our new OMD instantiation with the compression function of BLAKE2b.

Structure of the Paper. In Section 2 we introduce notations, recall the OMD authenticated encryption scheme and shortly describe basic facts regarding the hash functions SHA-512 and BLAKE2b. We propose a new instantiation of OMD and provide the reader with a short discussion regarding the security of the new instantiation in Section 3. In Section 4 we present the architecture of an online authenticated encryption system based on OMD as well as the results of our optimized implementations in hardware. Finally, we conclude in Section 5 and discuss future work ideas. We recall the pseudocode of OMD in Appendix A. The description of the compression functions of SHA-512 and BLAKE2b are given in Appendix B. We plot specific metrics of our implementations in Appendix C.

2 Preliminaries

Notations. During the following, $\|$ denotes string concatenation, $\oplus$ expresses the XOR operation and the notation 0^x refers to a string of x bits of zero. We denote by $x \leftarrow y$ the assignment of the value y to the variable x.

2.1 Offset Merkle Damgård

For recalling the main technical details of OMD we follow the descriptions of [2,7].

From a Compression Function to a Keyed Compression Function. Let function $F' : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$ be a compression function. F' can be turned into a *keyed compression function* F by using k bits of its b-bit input. More precisely, we may define $F_K(H, M) = F'(H, K\|M)$.

Specific Notations. Let function $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be a keyed compression function with $\mathcal{K} = \{0,1\}^k$ and $m \leq n$. The encryption tag will be denoted by Tag_e while the authentication tag is referred to as Tag_a. The final tag is denoted by Tag. We consider the length of Tag as being $\tau \in \{0, 1, \cdots, n\}$. Algorithms $\mathcal{E}$ (encryption) and $\mathcal{D}$ (decryption) can be called with arguments $K \in \mathcal{K}$, $N \in \{0,1\}^{\leq n-1}$ and $A, M, C \in \{0,1\}^*$, where A represents an associated data, M a message and C a ciphertext.

In the following, all OMD multiplications are performed in $\mathrm{GF}(2^n)$ and $\texttt{ntz}(i)$ denotes the number of trailing zeros (*i.e.* the number of rightmost bits that are

[4] Throughput to Area ratio

zero) in the binary representation of a positive integer i. Let N be the corresponding notation of a nonce[5]. We further denote by $\mathit{\Delta}_{N,i,j}$ and $\bar{\mathit{\Delta}}_{i,j}$ the masking values used in the OMD scheme (for processing the message and, respectively, the associated data). Let L_i be a sequence of additional values and $\ell_{\max}$ be the bound on the maximum number of m-bit blocks in any message that can be encrypted or decrypted.

Remark 1. The authors of [2,7] used the technique proposed in [10] to compute the masking values for assessing the security and efficiency requirements.

Initialization of OMD.

$$\begin{cases} \mathit{\Delta}_{N,0,0} & \leftarrow F_K(N\|10^{n-1-|N|}, 0^m) \\ \bar{\mathit{\Delta}}_{0,0} & \leftarrow 0^n \\ L_* & \leftarrow F_K(0^n, 0^m) \\ L[0] & \leftarrow 4L_* \\ L[i] & \leftarrow 2L[i-1] \text{ for } i \geq 1 \end{cases}$$

Remark 2. For a more efficient implementation, the values $L[i]$ can be preprocessed and stored in a table for $0 \leq i \leq \lceil \log_2(\ell_{\max}) \rceil$. The values $L[i]$ can also be computed on-the-fly for $i \geq 1$ in case of memory restrictions.

Masking Sequences for Processing the Message Equation (1) *and the Associated Data Equation* (2). For $i \geq 1$ we have that:

$$\begin{cases} \mathit{\Delta}_{N,i,0} \leftarrow \mathit{\Delta}_{N,i-1,0} \oplus L[\mathtt{ntz}(i)] \\ \mathit{\Delta}_{N,i,1} \leftarrow \mathit{\Delta}_{N,i,0} \oplus 2L_* \\ \mathit{\Delta}_{N,i,2} \leftarrow \mathit{\Delta}_{N,i,0} \oplus 3L_* \end{cases} \tag{1}$$

$$\begin{cases} \bar{\mathit{\Delta}}_{i,0} \leftarrow \bar{\mathit{\Delta}}_{i-1,0} \oplus L[\mathtt{ntz}(i)] \text{ for } i \geq 1 \\ \bar{\mathit{\Delta}}_{i,1} \leftarrow \bar{\mathit{\Delta}}_{i,0} \oplus L_* \text{ for } i \geq 0 \end{cases} \tag{2}$$

The OMD encryption algorithm generically instantiated with the compression function F keyed with K is presented in Figure 1. Note that for simplicity we depicted only the case of a message whose length is a multiple of the block length and an associated data whose length is a multiple of the input length. The cases in which padding is needed (both for messages and AD) are tackled in the pseudocode presented in Appendix A.

We recall the pseudocode of the four OMD sub-algorithms (*i.e.* INITIALIZE (K), HASH$_K(A)$, $\mathcal{E}_K(N, A, M)$ and $\mathcal{D}_K(N, A, C)$) in Appendix A.

[5] number used only once

Nonces. The security proofs of OMD hold as long as the principle of non-repeating nonces is respected (uniqueness criterion). In standard encryption applications the nonce is usually a counter sent over the communication channel along with the authenticated and encrypted message. In practice, the nonce has to be unique during an encryption session (*i.e* during the lifetime of a session key). The nonce is needed both for encryption and decryption and can be communicated in clear between the two corresponding parties. The uniqueness criterion is encryption related in the sense that the user wishing to transmit a message to another user is responsible of generating suitable[6] nonces.

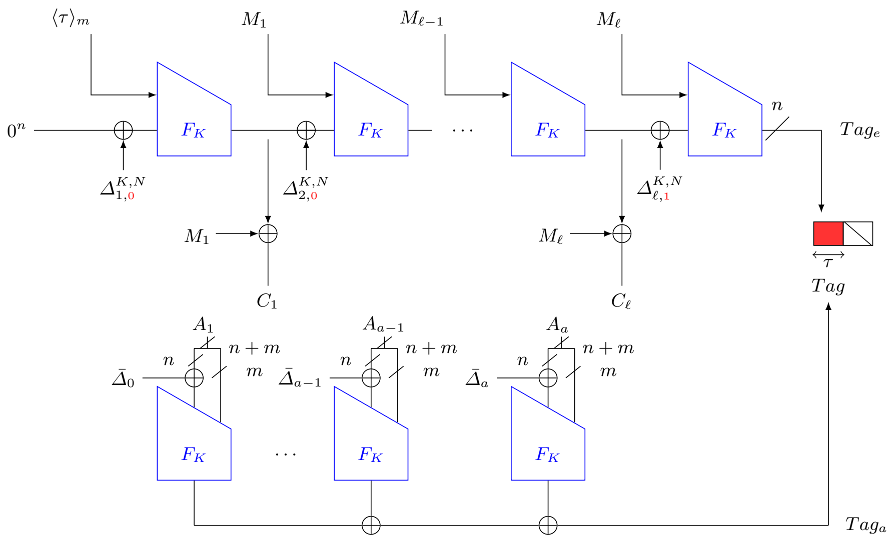

Fig. 1. OMD in the Case of a Message whose Length is a Multiple of the Block Length and an Associated Data whose Length is a Multiple of the Input Length.

2.2 The Hash Functions SHA-512 and BLAKE2b

The SHA-2 family of hash functions [13] is still one of the *de facto* standards when it comes to hash functions. Even though SHA-3 is the latest member of the SHA (Secure Hash Family) family of functions, SHA-2 still stands from the security point of view.

The hash function BLAKE2 [6] is a modified version of BLAKE [6] which is a finalist of the SHA-3 cryptographic competition. BLAKE2 was constructed to supersede BLAKE's efficiency (*i.e.* optimize it for modern applications). As BLAKE2 is really appealing developers it has already been used in several

[6] from a security perspective

projects, including the widely adopted WinRAR archiving utility and the memory hard key derivation function Argon2 (the winner of the Password Hashing Competition [3]).

We provide the reader with technical details of sha-512 and blake2b which are relevant for our paper in Appendix B.

3 A New OMD Instantiation. Security Aspects

As already mentioned, OMD is an authenticated encryption scheme based on compression functions. For the purpose of our paper we selected the secondary CAESAR recommendation of the OMD scheme to be implemented in hardware.

Moreover, we propose and analyze a new instantiation of OMD instead of the two original ones. As stated in the previous sections, we chose the compression function of BLAKE2b.

We further denote by sha-512 the compression function of SHA-512 and by blake2b the compression function of BLAKE2b. Furthermore, we denote by OMD-sha-512 and by OMD-blake2b the OMD instantiations based on the compression functions of SHA-512, respectively BLAKE2b.

3.1 Security Analysis

As OMD is a nonce-based AEAD scheme, its authors aimed at achieving the security notions for AEAD schemes as detailed in [16]. The security of the OMD scheme as well as the security of its primary and secondary instantiations (*i.e.* OMD-sha-256 and OMD-sha-512) are discussed in an extensive manner in [2,7]. It is straightforward that the security proofs still hold in the case of our proposed instantiation, *i.e.* OMD-blake2b.

4 Speeding-Up OMD in Hardware. Implementation Trade-Offs

In this section we present the architecture of an online authenticated encryption system based on specific OMD instantiations, while our main focus is on speeding-up OMD in hardware. We start by giving the general architecture and continue with implementation details of OMD and particular instantiations of it in Section 4.1 (OMD-sha-512 and OMD-blake2b). In order to underline our speed-ups, we provide the reader with comparison results between our implementations and other related works in Section 4.2.

Target FPGA Platform. The hardware implementation of the OMD scheme was realized using register transfer-level (RTL) design methodology. We adopted the VHDL as our preferred hardware description language (HDL) in order to implement the necessary hardware components for the FPGA circuit design. We opted for Virtex UltraScale+ VCU118[7] which is an effective platform from the point of view of its resources (I/O pins, QSFP28 Interfaces, high on-chip memory density, etc.). Thus, the platform we chose is a very good option for the future development of an online system.

All the development was done using the Xilinx Vivado Design Suite 2019.1 and it involved the following steps:

1. Designing the top view architecture of the whole system and defining the input/output ports of the main modules;
2. Writing the VHDL code for the previous specified modules;
3. Writing simulation testbenches in order to validate the functionality of the modules;
4. Synthesizing the design and checking for any possible errors;
5. Implementing the design;
6. Analyzing the timing requirements and the resources.

4.1 The Architecture of a Real World Authenticated Encryption System

The design of our proposed system is composed of a Top Module (shown in Figure 2) that contains all the components of the circuit which is described as:

– **PTXT IF**: this block represents the plaintext interface and it is used to receive and send data packets in the trusted area of a network;
– **Receive**: transfers the data from the receive PTXT IF to the FIFO PTXT block;
– **FIFO PTXT**: implements a FIFO module for plaintext packets storage; it is also used for Clock Domain Crossing (CDC);
– **PUT to ENC**: reads the data from FIFO PTXT and prepare the packets for the encryption block;
– **ENCRYPT**: encrypts the data using the OMD scheme depicted in Figure 3;
– **GET from ENC**: takes the encrypted blocks and creates the encrypted packet which is then written in the FIFO ENC;
– **FIFO ENC**: implements a FIFO module for ciphertext packets storage; it is also used for Clock Domain Crossing (CDC);
– **Send**: transfers the data from the FIFO ENC block to the transmit CTXT IF;
– **CTXT IF**: this block represents the ciphertext interface and it is used to receive and send data packets in the untrusted area of a network;
– **KS GEN**: this block is used to generate a common session key between two communicating parties; it also computes the L and Tag_a values which are used by the ENCRYPT/DECRYPT blocks;
– **NONCE GEN**: this block is used to generate nonces which are unique per session;

[7] xcvu9p-flga2104-2L-e

- **DECRYPT**: this block is identical to ENCRYPT, except an additional tag verification that is done at the decryption of the message;
- the rest of the blocks complete the scheme in a symmetric manner, offering similar functionalities as the already described ones;

We continue to focus on the main blocks of the system (ENCRYPT and DECRYPT) which includes the OMD algorithm. The block diagram illustrated in Figure 3 consists of modules which are further categorized as Datapath or Controller modules. Datapath describes how the data moves through the system at register level, leveraging the parallel nature of the FPGA circuits. The Controller consists of a Finite State Machine (FSM) which runs sequentially providing the decision logic of the system. In order to simplify the illustrated design, we presented in Figure 3 only the Datapath logic.

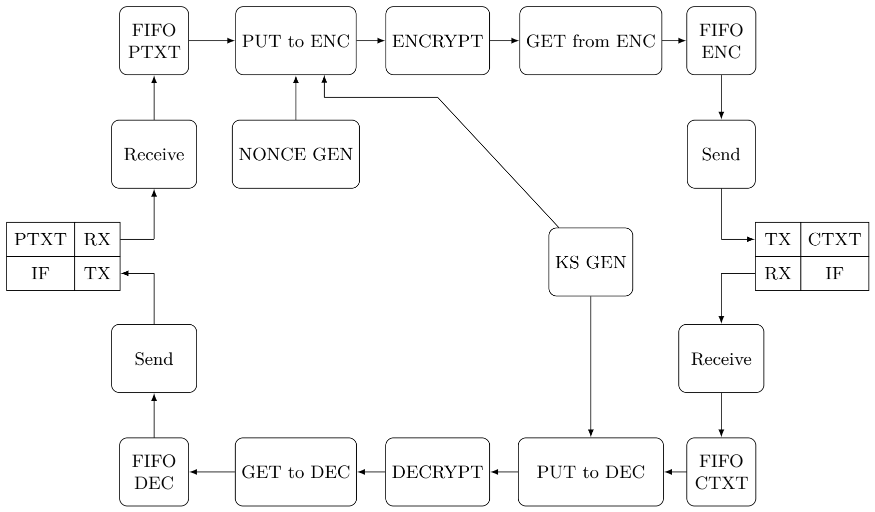

Fig. 2. A General Architecture of an Online System Using OMD

In the case of the OMD algorithm the encryption of the current block is dependent on the previous one (*i.e.* the scheme is sequential), thus we can only use an iterative implementation. In this case, the pipelined implementation is not feasible. The original OMD scheme is reduced to a single F_k block where the inputs and outputs are managed by the controller in the following way: the inputs are multiplexed with different data paths and the outputs are used to compute the ciphertext and Tag_e or to be fed back into the F_k block. In Figure 3, the Datapath is described in a simplified diagram which has the following main parts: ① data multiplexing modules (for changing the datapath to the inputs of the F_k block), ② registers (for storing temporary data) and ③ RAM memory (for storing the computed values).

The input of the ENCRYPT block consists of:

– Message M (divided into blocks of length 512 bits);
– Secret key K (of length 512 bits);
– Nonce N (of length 256 bits which are provided by the nonce generator and are unique per session);
– Two precomputed values L and Tag_a (calculated using the KS GEN block).

We chose to calculate Tag_a and L only once per session as they do not depend on the message or on the nonce. This fact is reflected in the number of LUTs and the TP/A values presented in Table 1. Thus, the values written inside the parentheses denote the number of LUTs and the TP/A values without the need of calculating Tag_a and L.

The output of the DECRYPT block consists of ciphertext C (divided into blocks of 512 bits) and Tag (of length 512 bits).

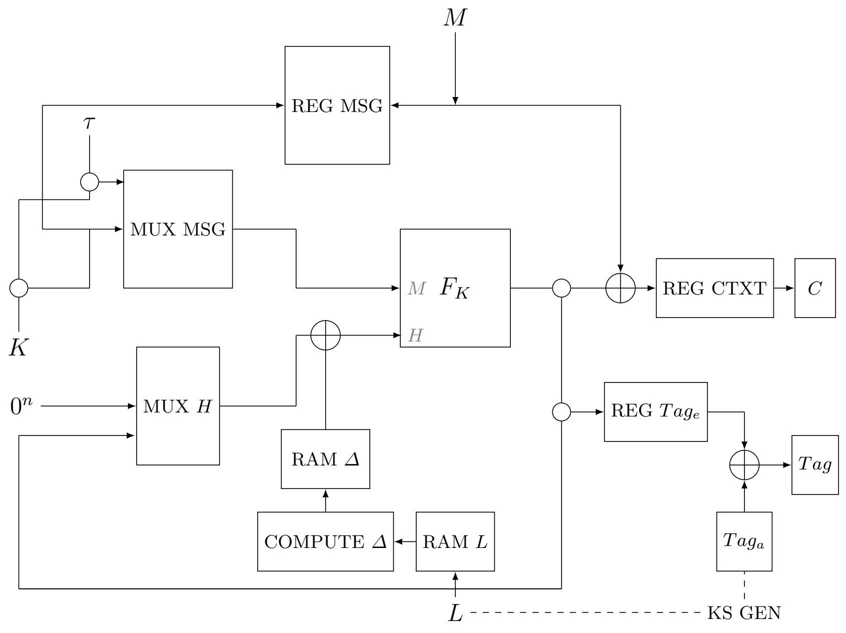

Fig. 3. OMD Encrypt Block Diagram.

The results of the OMD RTL implementations are described next. Implementation statistics in the Virtex UltraScale+ are shown in Table 1. We provide metrics regarding Throughput, Area and Throughput-to-Area (TP/A) ratio, LUTs, LUT RAM and Frequency. Throughput is defined in terms of 10^9 bits/second (Gbps) and area is defined in terms of LUTs (LookUp Tables). We also show the number of clock cycles used to calculate the throughput for long messages.

The encryption process of OMD is splitted in three phases in the case of our hardware implementation: ① Setup (the initialization phase), ② Message (the processing time of all the n blocks of the message) and ③ Tag (the final phase in which the tag is computed).

Metric	**OMD-sha-512**	**OMD-blake2b**
Setup (CLK)	116	48
Message (CLK)	$93 \cdot n$	$25 \cdot n$
Tag (CLK)	91	23
Frequency (MHz)	250	125
Throughput (Gbps)	1.3	2.56
LUTs	7187 (3451)	18907 (14875)
LUT RAM	3736 (0)	3736 (296)
Throughput/Area	0.18 (0.3736)	0.125 (0.159)

Table 1. Implementation Metrics for OMD-sha-512 and OMD-blake2b.

Open Source Implementation. The VHDL source code of our OMD optimized implementations may be found at [4].

4.2 Results Comparison

Comparison between OMD-sha-256 and OMD-sha-512. In [8] an implementation of the primary OMD instantiation (OMD-sha-256) is discussed. Given that we focused on efficiently implementing the secondary OMD instantiation (OMD-sha-512) we need to abstractly scale the results of [8] for a fair comparison taking into account the natural differences that appear because we replaced sha-256 with sha-512. Also, we note that there is a difference in terms of FPGA target platform in the sense that the authors of [8] use a Virtex 7 FPGA platform. We stress that our superior implementation results are not only due to the newer FPGA platform we used, but also due to the implementation techniques we employed. Moreover, we underline that in [8] only "long messages" (according to the authors) are considered for computing the implementation metrics. We believe that the notion "long message" should be clearly defined in order to obtain accurate results. All in all, we report the next differences between the two previously mentioned implementations:

- In terms of Throughput we obtained 1.3 Gbps as opposed to 1.071 Gbps;
- In terms of Throughput-to-Area we obtained 0.18 (0.3736) as opposed to 0.228;
- In terms of Frequency we obtained 250 MHz as opposed to 276 Mhz;
- In terms of LUTs we obtained 7187 (3451) as opposed to 4701.

As a conclusion, even though OMD-sha-512's security is higher than OMD-sha-256's and we used a key length of 512 bits as opposed to a key length of

128 bits, our throughput supersedes the throughput reported in [8]. This is an important feature in real world applications which need to transfer data at a high rate. Although the parameters we used for implementing OMD-sha-512 are at least double as compared to the ones used in [8], the TP/A, the Frequency and the number of LUTs are way smaller than the double of the values reported in [8]. We also have to mention the fact that even though we used more LUTs, we utilized at most 3% of the platform's available resources. Furthermore, the authors of [8] do not implement a mechanism similar to ours for computing Tag_a and L only once per session.

Comparison between OMD-sha-512 and OMD-blake2b. We report the next differences between our implementations of OMD-sha-512 and OMD-blake2b:

- In terms of Setup we obtained 116 clock cycles as opposed to 48;
- In terms of Message we obtained $93 \cdot n$ clock cycles as opposed to $25 \cdot n$;
- In terms of Tag we obtained 91 clock cycles as opposed 23;
- In terms of Throughput we obtained 1.3 Gbps as opposed to 2.56 Gbps;
- In terms of Throughput-to-Area we obtained 0.18 (0.3736) as opposed to 0.125 (0.159);
- In terms of Frequency we obtained 250 MHz as opposed to 125 MHz;
- In terms of LUTs we obtained 7187 (3451) as opposed to 18907 (14875).

As a conclusion, the Throughput of OMD-blake2b is higher than OMD-sha-512 due to the following facts: ① the blake2b compression function has only 12 rounds as opposed to sha-512 which has 80 rounds and ② each round of both blake2b and sha-512 compression functions takes only one clock cycle. Concerning the number of LUTs we have to mention that OMD-sha-512 is a better option for Area constrained platforms, while OMD-blake2b is a better option for real world applications which need to transfer data at a high rate. We also have to point out the differences between the frequency values in OMD-sha-512 (250 MHz) and OMD-blake2b (125 MHz), which are due to the more complex structure of the blake2b compression function.

5 Conclusions and Future Work

We proposed a new OMD instantiation (OMD-blake2b) and showed how to use it as the main cryptographic primitive of a real world authenticated encryption system. We presented the results of our optimized implementations in hardware and provided the reader with a security analysis of our proposed instantiation.

Future Work. After the original OMD scheme was submitted to CAESAR, different flavours of it were proposed in the literature: two nonce misuse-resistant variants [14] and pure OMD (p-OMD) [15], a more efficient OMD version (*i.e.* the associated data is processed almost for free). Besides inheriting all the security features of OMD, the authors claim authenticity against nonce-misusing adversaries. Note that an important update regarding the security of p-OMD is presented in [5]: it is shown that p-OMD does not actually achieve authenticity against misuse-resistant adversaries. The attack is strictly specific to p-OMD and does not invalidate its main result on nonce-respecting adversaries[8]. Thus, from both the diversity and efficiency points of view, we believe that a straightforward future approach is to provide hardware implementation metrics for all previously mentioned OMD variants.

Another interesting research direction would be to analyze the security of our proposed OMD optimized implementations against physical attacks. Additionally, suitable countermeasures against such attacks are to be considered as future work (*e.g.* masking techniques).

6 Acknowledgments

The authors would like to thank Traian Neacşa and George Teşeleanu for their helpful comments.

References

1. CAESAR. https://competitions.cr.yp.to/caesar.html
2. OMDv2 CAESAR Submission. https://competitions.cr.yp.to/round2/omdv20c.pdf
3. Password Hashing Competition. https://password-hashing.net
4. Source Code. https://github.com/megastefan22/OMD
5. Ashur, T., Mennink, B.: Trivial Nonce-Misusing Attack on Pure OMD. IACR Cryptology ePrint Archive (2015)
6. Aumasson, J.P., Meier, W., Phan, R., Henzen, L.: The Hash Function BLAKE. Springer (2014)
7. Cogliani, S., Maimuţ, D., Naccache, D., Portella do Canto, R., Reyhanitabar, R., Vaudenay, S., Vizár, D.: OMD: A Compression Function Mode of Operation for Authenticated Encryption. In: Selected Areas in Cryptography - SAC'14. Lecture Notes in Computer Science, vol. 8781, pp. 112–128. Springer (2014)
8. Diehl, W., Gaj, K.: RTL Implementations and FPGA Benchmarking of Selected CAESAR Round Two Authenticated Ciphers. Microprocessors and Microsystems (2017), https://www.sciencedirect.com/science/article/abs/pii/S0141933117300352
9. Homsirikamol, E., Rogawski, M., Gaj, K.: Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. IACR Cryptology ePrint Archive (2010), http://eprint.iacr.org/2010/445
10. Krovetz, T., Rogaway, P.: The Software Performance of Authenticated-Encryption Modes. In: FSE 2011. Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer (2011)
11. Maimuţ, D.: Authentication and Encryption Protocols: Design, Attacks and Algorithmic Tools. Ph.D. thesis, École normale supérieure (2015)
12. Maimuţ, D., Reyhanitabar, R.: Authenticated Encryption: Toward Next-Generation Algorithms. IEEE Security & Privacy **12**(2), 70–72 (2014)

[8] Moreover, the attack does not apply to the OMD CAESAR submission and to the misuse-resistant variants of [14].

13. National Institute of Standards and Technology: FIPS PUB 180-4: Secure Hash Standard. NIST (aug 2015)
14. Reyhanitabar, R., Vaudenay, S., Vizár, D.: Misuse-Resistant Variants of the OMD Authenticated Encryption Mode. In: Provable Security - ProvSec'14. Lecture Notes in Computer Science, vol. 8782, pp. 55–70. Springer (2014)
15. Reyhanitabar, R., Vaudenay, S., Vizár, D.: Boosting OMD for Almost Free Authentication of Associated Data. In: Fast Software Encryption - FSE'15. Lecture Notes in Computer Science, vol. 9054, pp. 411–427. Springer (2015)
16. Rogaway, P.: Authenticated-encryption with associated-data. In: CCS'02. pp. 98–107. ACM (2002)
17. Rogaway, P.: Nonce-Based Symmetric Encryption. In: FSE'04. Lecture Notes in Computer Science, vol. 3017, pp. 348–359. Springer (2004)

A OMD Pseudocode

Algorithm 1: Initialize (K)

```
$L_* \leftarrow F_K(0^n, 0^m)$
$L[0] \leftarrow 4.L_*$                                  ▷ $2.(2.L_*)$, doubling in $\mathrm{GF}(2^n)$
for $i \leftarrow 1$ to $\lceil \log_2(\ell_{\max}) \rceil$ do
    $L[i] = 2.L[i-1]$                                     ▷ doubling in $\mathrm{GF}(2^n)$
end
return
```

Algorithm 2: $\textsc{Hash}_K(A)$

```
$b \leftarrow n + m$
$A_1 || A_2 || \cdots || A_{\ell-1} || A_\ell \xleftarrow{b} A$, where $|A_i| = b$ for $1 \le i \le \ell - 1$ and $|A_\ell| \le b$
$\mathrm{Tag}_a \leftarrow 0^n$
$\Delta \leftarrow 0^n$
for $i \leftarrow 1$ to $\ell - 1$ do
    $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(i)]$
    $\mathsf{Left} \leftarrow A_i[b-1, \cdots, m]$
    $\mathsf{Right} \leftarrow A_i[m-1, \cdots, 0]$
    $\mathrm{Tag}_a \leftarrow \mathrm{Tag}_a \oplus F_K(\mathsf{Left} \oplus \Delta, \mathsf{Right})$
end
if $|A_\ell| = b$ then
    $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(\ell)]$
    $\mathsf{Left} \leftarrow A_\ell[b-1, \cdots, m]$
    $\mathsf{Right} \leftarrow A_\ell[m-1, \cdots, 0]$
    $\mathrm{Tag}_a \leftarrow \mathrm{Tag}_a \oplus F_K(\mathsf{Left} \oplus \Delta, \mathsf{Right})$
end
else
    $\Delta \leftarrow \Delta \oplus L_*$
    $\mathsf{Left} \leftarrow A_\ell || 10^{b-|A_\ell|-1}[b-1, \cdots, m]$
    $\mathsf{Right} \leftarrow A_\ell || 10^{b-|A_\ell|-1}[m-1, \cdots, 0]$
    $\mathrm{Tag}_a \leftarrow \mathrm{Tag}_a \oplus F_K(\mathsf{Left} \oplus \Delta, \mathsf{Right})$
end
return $\mathrm{Tag}_a$
```

Algorithm 3: $\mathcal{E}_K(N, A, M)$

```
if $|N| > n-1$ then
    return
end
$\perp M_1||M_2||\cdots||M_{\ell-1}||M_\ell \xleftarrow{m} M$, where $|M_i| = m$ for $1 \leq i \leq \ell-1$ and $|M_\ell| \leq m$
$\Delta \leftarrow F_K(N||10^{n-1-|N|}, 0^m)$                    ▷ initialize $\Delta_{N,0,0}$
$H \leftarrow 0^n$
$\Delta \leftarrow \Delta \oplus L[0]$                    ▷ compute $\Delta_{N,1,0}$
$H \leftarrow F_K(H \oplus \Delta, \langle\tau\rangle_m)$
for $i \leftarrow 1$ to $\ell - 1$ do
    $C_i \leftarrow H \oplus M_i$
    $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(i+1)]$
    $H \leftarrow F_K(H \oplus \Delta, M_i)$
end
$C_\ell \leftarrow H \oplus M_\ell$ if $|M_\ell| = m$ then
    $\Delta \leftarrow \Delta \oplus 2.L_*$
    $\mathrm{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell)$
end
else
    if $|M_\ell| \neq 0$ then
        $\Delta \leftarrow \Delta \oplus 3.L_*$
        $\mathrm{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell||10^{m-|M_\ell|-1})$
    end
end
else
    $\mathrm{Tag}_e \leftarrow H$
end
$\mathrm{Tag}_a \leftarrow \textsc{Hash}_K(A)$
$\mathrm{Tag} \leftarrow (\mathrm{Tag}_e \oplus \mathsf{Tag}_a)[n-1, \cdots, n-\tau]$
$C \leftarrow C_1||C_2||\cdots||C_\ell||\mathsf{Tag}$ return $C$
```

Algorithm 4: $\mathcal{D}_K(N, A, \mathbb{C})$

```
if $|N| > n-1$ or $|C| < \tau$ then
    return $\perp$
end
$C_1||C_2||\cdots||C_{\ell-1}||C_\ell||\mathsf{Tag} \xleftarrow{m} C$, where $|C_i| = m$ for $1 \leq i \leq \ell - 1$, $|C_\ell| \leq m$
  and $|\mathsf{Tag}| = \tau$
$\Delta \leftarrow F_K(N||10^{n-1-|N|}, 0^m)$                                    ▷ initialize $\Delta_{N,0,0}$
$H \leftarrow 0^n$
$\Delta \leftarrow \Delta \oplus L[0]$                                               ▷ compute $\Delta_{N,1,0}$
$H \leftarrow F_K(H \oplus \Delta, \langle\tau\rangle_m)$
for $i \leftarrow 1$ to $\ell - 1$ do
    $M_i \leftarrow H \oplus C_i$
    $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(i+1)]$
    $H \leftarrow F_K(H \oplus \Delta, M_i)$
end
$M_\ell \leftarrow H \oplus C_\ell$
if $|C_\ell| = m$ then
    $\Delta \leftarrow \Delta \oplus 2.L_*$
    $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell)$
end
else
    if $|C_\ell| \neq 0$ then
        $\Delta \leftarrow \Delta \oplus 3.L_*$
        $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell||10^{m-|M_\ell|-1})$
    end
end
else
    $\mathsf{Tag}_e \leftarrow H$
end
$\mathsf{Tag}_a \leftarrow \textsc{Hash}_K(A)$
$\mathsf{Tag}' \leftarrow (\mathsf{Tag}_e \oplus \mathsf{Tag}_a)[n-1, \cdots, n-\tau]$
if $\mathit{Tag}' = \mathit{Tag}$ then
    return $M \leftarrow M_1||M_2||\cdots||M_\ell$
end
else
    return $\perp$
end
```

B The sha-512 and blake2b Compression Functions

B.1 Preliminaries

In the following, by "word" we mean a group of $w = 64$ bits. Namely, in sha-512 each word is a 64-bit string.

$ROTR^n(x)$ and $SHR^n(x)$: Let x be a w-bit word and n an integer with $0 \leq n < w$. The *rotate right* (circular right shift) operation is defined by $ROTR^n(x) = (x \gg n) \vee (x \ll w - n)$. The *right shift* operation is defined by $SHR^n(x) = (x \gg n)$.

Choice and Majority Functions. The *choice function* and *majority function* (also called the median operator) functions can be defined as follows:

$$Ch : \left| \begin{array}{r} \{0,1\}^m \times \{0,1\}^m \times \{0,1\}^m \longrightarrow \{0,1\}^m \\ x, y, z \longmapsto (x \wedge y) \oplus (\neg x \wedge z) \end{array} \right.$$

$$Maj : \left| \begin{array}{r} \{0,1\}^m \times \{0,1\}^m \times \{0,1\}^m \longrightarrow \{0,1\}^m \\ x, y, z \longmapsto (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \end{array} \right.$$

B.2 The sha-512 Compression Function

Sigma Functions. The functions $\Sigma_0^{\{512\}}$ and $\Sigma_1^{\{512\}}$ are defined as follows:

$$\Sigma_0^{\{512\}} \left| \begin{array}{r} \{0,1\}^{64} \longrightarrow \{0,1\}^{64} \\ x \longmapsto ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \end{array} \right.$$

$$\Sigma_1^{\{512\}} \left| \begin{array}{r} \{0,1\}^{64} \longrightarrow \{0,1\}^{64} \\ x \longmapsto ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \end{array} \right.$$

The $\sigma_0^{\{512\}}$ and $\sigma_1^{\{512\}}$ functions are defined as follows:

$$\sigma_0^{\{512\}} \left| \begin{array}{r} \{0,1\}^{64} \longrightarrow \{0,1\}^{64} \\ x \longmapsto ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \end{array} \right.$$

$$\sigma_0^{\{512\}} \left| \begin{array}{r} \{0,1\}^{64} \longrightarrow \{0,1\}^{64} \\ x \longmapsto SHR^{19}(x) \oplus SHR^{61}(x) \oplus SHR^{6}(x) \end{array} \right.$$

The Process. The sha-512 compression function is defined as:

$$sha - 512 \left| \begin{array}{r} \{0,1\}^{512} \times \{0,1\}^{1024} \longrightarrow \{0,1\}^{512} \\ H, M \longmapsto D \end{array} \right.$$

Let M be the 1024-bit *message input* and H the 512-bit *hash input* (chaining input). These two inputs are represented respectively by an array of 16 64-bit words $M_0 \| \cdots \| M_{15}$, and an array of 8 64-bit words $H_0 \| \cdots \| H_7$. The 512-bit output value C is also represented as an array of 8 64-bit words $D_0 \| \cdots \| D_7$.

Let H be the 512-bit *hash input* (chaining input) and M be the 1024-bit *message input.* These two inputs are represented respectively by an array of 8 64-bit words $H_0 \| \cdots \| H_7$ (see Table 2) and an array of 16 64-bit words $M_0 \| \cdots \| M_{15}$. The 512-bit output value D is also represented as an array of 8 64-bit words $D_0 \| \cdots \| D_7$.

$$\begin{aligned}
H_0 &= \texttt{6a09e667f3bcc908}\\
H_1 &= \texttt{bb67ae8584caa73b}\\
H_2 &= \texttt{3c6ef372fe94f82b}\\
H_3 &= \texttt{a54ff53a5f1d36f1}\\
H_4 &= \texttt{510e527fade682d1}\\
H_5 &= \texttt{9b05688c2b3e6c1f}\\
H_6 &= \texttt{1f83d9abfb41bd6b}\\
H_7 &= \texttt{5be0cd19137e2179}
\end{aligned}$$

Table 2. sha-512 Initial Values

During the process of compression, a sequence of 80 constant 64-bit words $K_0^{\{512\}}$, ..., $K_{79}^{\{512\}}$ is used. These 64-bit words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. In hex, these constant words are given in Table 3 (from left to right).

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

Table 3. sha-512 Constants

We further provide the reader with the description of the sha-512 compression function. The addition $(+)$ is performed modulo 2^{64}.

1. Preparing the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t, & 0 \leq t \leq 15 \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16}, & 16 \leq t \leq 79 \end{cases}$$

2. Initialize the eight working variables, a, b, c, d, e, f, g and h with the hash input value H:

$a = H_0 \qquad b = H_1 \qquad c = H_2 \qquad d = H_3$
$e = H_4 \qquad f = H_5 \qquad g = H_6 \qquad h = H_7$

3. For $t = 0$ to 79, do:
$\{$
$T_1 \ = \ h + \Sigma_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t$
$T_2 = \Sigma_0^{\{512\}}(a) + Maj(a, b, c)$
$h = g \qquad g = f \qquad f = e \qquad e = d + T_1$
$d = c \qquad c = b \qquad b = a \qquad a = T_1 + T_2$
$\}$
4. Computing the 512-bit output (hash) value $C = C_0 \cdots C_7$ as:
$C_0 = a + H_0 \qquad C_1 = b + H_1 \qquad C_2 = c + H_2 \qquad C_3 = d + H_3$
$C_4 = e + H_4 \qquad C_5 = f + H_5 \qquad C_6 = g + H_6 \qquad C_7 = h + H_7$

B.3 The blake2b Compression Function

The initial values H of blake2b was chosen precisely as the ones for SHA-512 (given in Table 2). These values "were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers", according to [13].

Thus, the compression function blake2b takes as input:

$H \ = H_0 \| H_1 \| \dots \| H_7$	(of length 512 bits)
$M = M_0 \| M_1 \| \dots \| M_7$	(of length 1024 bits)
$T \ = T_0 \| T_1$	(of length 128 bits)
$F \ = F_0 \| F_1$	(of length 128 bits)

$$\begin{pmatrix} \nu_0 & \nu_1 & \nu_2 & \nu_3 \\ \nu_4 & \nu_5 & \nu_6 & \nu_7 \\ \nu_8 & \nu_9 & \nu_{10} & \nu_{11} \\ \nu_{12} & \nu_{13} & \nu_{14} & \nu_{15} \end{pmatrix} := \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ H_0 & H_1 & H_2 & H_3 \\ T_0 \oplus H_4 & T_1 \oplus H_5 & F_0 \oplus H_6 & F_1 \oplus H_7 \end{pmatrix}$$

σ_0 :	[0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15]
σ_1 :	[14,	10,	4,	8,	9,	15,	13,	6,	1,	12,	0,	2,	11,	7,	5,	3]
σ_2 :	[11,	8,	12,	0,	5,	2,	15,	13,	10,	14,	3,	6,	7,	1,	9,	4]
σ_3 :	[7,	9,	3,	1,	13,	12,	11,	14,	2,	6,	5,	10,	4,	0,	15,	8]
σ_4 :	[9,	0,	5,	7,	2,	4,	10,	15,	14,	1,	11,	12,	6,	8,	3,	13]
σ_5 :	[2,	12,	6,	10,	0,	11,	8,	3,	4,	13,	7,	5,	15,	14,	1,	9]
σ_6 :	[12,	5,	1,	15,	14,	13,	4,	10,	0,	7,	6,	3,	9,	2,	8,	11]
σ_7 :	[13,	11,	7,	14,	12,	1,	3,	9,	5,	0,	15,	4,	8,	6,	2,	10]
σ_8 :	[6,	15,	14,	9,	11,	3,	0,	8,	12,	2,	13,	7,	1,	4,	10,	5]
σ_9 :	[10,	2,	8,	4,	7,	6,	1,	5,	15,	11,	9,	14,	3,	12,	13,	0]

Table 4. Permutations of blake2b

Let the round permutations σ_r be in accordance with Table 4, where $r = \overline{0, 9}$. Note that for rounds $r \geq 10$ the permutation used is $\sigma_{r \bmod 10}$. The core function G of blake2b is defined as follows:

$a := \ a + b + m_{\sigma_r(2i)} \qquad d := \ ROTR^{32}(d \oplus a) \qquad c := \ c + d \qquad b := \ ROTR^{24}(b \oplus c)$
$a := \ a + b + m_{\sigma_r(2i+1)} \qquad d := \ ROTR^{16}(d \oplus a) \qquad c := \ c + d \qquad b := \ ROTR^{63}(b \oplus c)$

C Explicit Performance Metrics

The main performance metrics which are used in our work are throughput, area and throughput to area ratio (Tp/A). They are presented in Figures 4 to 6 in comparison with the block size of a message (which is represented on the x axis in all plots). The block size n goes from 64 bytes to 9600 bytes (we chose these values in order to meet the requirements of an online system). Both OMD-sha-512 and OMD-blake2b instantiations are taken into consideration.

All formulas used to generate the plots are based on the metrics described in Table 1. We recall that, during the following, ① *Setup* represents the number of clock cycles necessary in the initialization phase, ② *Message* refers to the number of clock cycles necessary to process a message composed of n blocks, ③ *Tag* represents the number of clock cycles necessary to calculate Tag and ④ *Frequency* refers to the frequency of the FPGA circuit.

In Figure 4 we use latency as the parameter represented by the y axis. We computed latency by the following formula:

$$\text{Latency} = \text{CLK} \cdot (\text{Setup} + n \cdot \text{Message} + \text{Tag}) \cdot 1/\text{Frequency}.$$

We computed the throughput by applying the following formula:

$$\text{Throughput} = \frac{n \cdot 64 \cdot 8 \cdot \text{Frequency}}{\text{CLK} \cdot (\text{Setup} + n \cdot \text{Message} + \text{Tag})}.$$

The computation of Tp/A is straightforward.

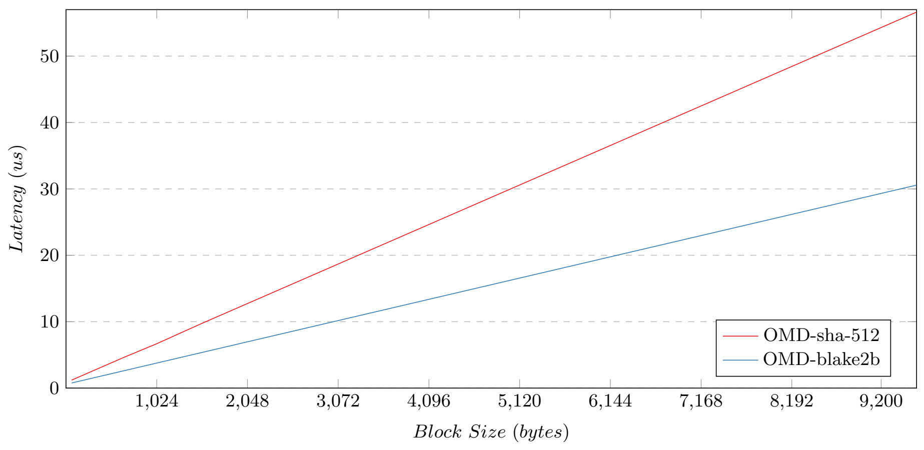

Fig. 4. Latency vs. Block Size

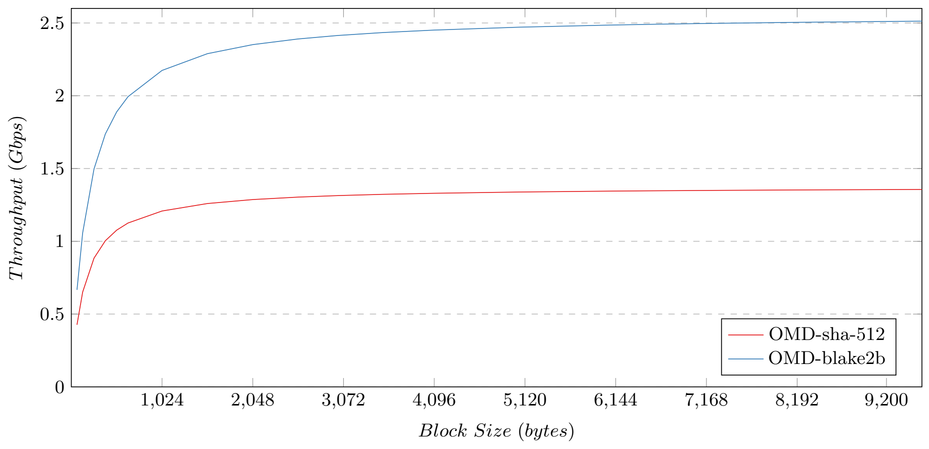

Fig. 5. Throughput vs. Block Size

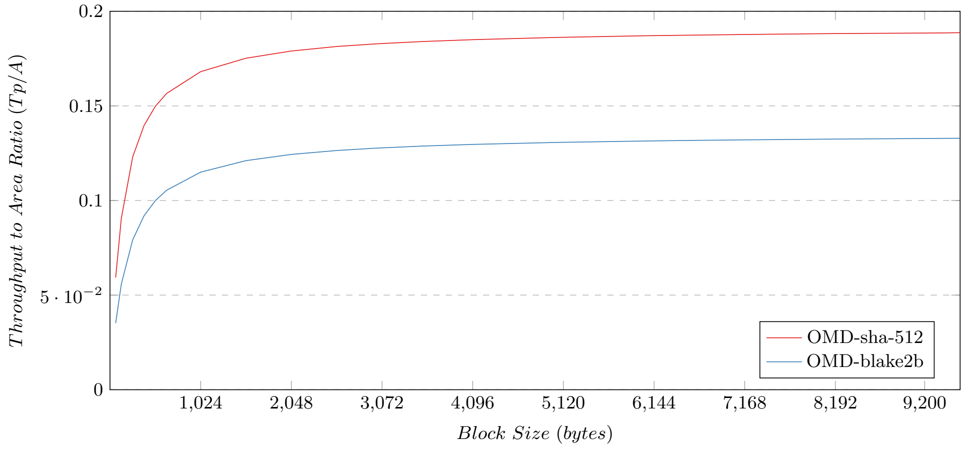

Fig. 6. Throughput to Area Ratio (Tp/A) vs. Block Size