Offset Merkle-Damgård (OMD) version 2.0
A CAESAR Proposal

Simon Cogliani[1], Diana Maimut[1], David Naccache[1], Rodrigo Portella[2], Reza Reyhanitabar[3], Serge Vaudenay[3], and Damian Vizár[3]

[1]ENS, France
[2]Université Paris II - Panthéon-Assas, France
[3]EPFL, Switzerland

Contact: `reza.reyhanitabar@epfl.ch`

August 25, 2015

Summary

This document describes a nonce-based authenticated encryption with associated data (AEAD) scheme proposed to the CAESAR competition. Our proposal, called Offset Merkle-Damgård (OMD for short), is a mode of operation for a keyed compression function. If the compression function at hand does not have a dedicated-key input then it must be first keyed by some conventional method, e.g. prepending the key to the message in the input of the compression function.

Almost all popular AEAD schemes in the literature have used blockciphers or (more recently) random permutations as their underlying primitives. But, we note that compression functions are also among the most widely-used and analyzed cryptographic primitives. We have a rich source of secure compression functions thanks to more than two decades of public research and standardization activities on hash functions. This motivates one to think about building an AEAD scheme from a compression function. However, at first glance, it might seem that the efficiency of a compression function based AEAD scheme would be, in general, lower than popular blockchipher-based or permutation-based AEAD schemes. This efficiency concern has been perhaps the reason for less interest on compression function based designs for authenticate encryption. But, the recent announcement by Intel® in July 2013 about introduction of new instructions, supporting performance acceleration of the Secure Hash Algorithm (SHA) on Intel® Architecture processors (in particular, for SHA-1 and SHA-256), makes one rethink about efficiency of compression function based AEAD design.

OMD takes advantage of the aforementioned facts about compression functions and provides a scheme whose security is proven based on well-established security properties of the underlying compression function and has promising performance thanks to the new Intel® instructions supporting the SHA on Intel® architecture processors. As specific instantiations of the OMD mode, we recommend two specific compression functions to be keyed and used in OMD, namely, the compression functions of the standard SHA-256 and SHA-512 hash functions. OMD parametrized with these two compression functions is called OMD-sha256 and OMD-sha512, respectively. The former is intended for 32-bit implementations and is our main recommended cipher for CAESAR, while the latter could be used specifically for 64-bit machines.

OMD achieves nearly optimal performance in terms of number of compression function calls that one can expect from any AEAD scheme *solely* using a compression function. OMD has several attractive features: (1)

unlike the permutation-based schemes whose security relies on *idealistic* assumptions about their underlying permutation, the security of OMD is proved in the standard model based on merely the classical PRF assumption on the compression function, (2) one can easily get a high quantitative level of security using OMD with the compression function of a standard hash function with a large hash size (e.g. 256 bits or 512 bits), (3) the only operations that OMD needs in addition to its core compression function are the basic operations of bitwise xoring two binary strings and shifting a binary string to the left, (4) selecting the core compression function to be that of SHA-256 the scheme can take advantage of the new Intel® instructions for a highly efficient implementation.

Contents

1 Introduction **4**

2 Preliminaries **5**

3 Security Goals **8**
3.1 Quantitative Security Level of OMD-sha256 . . . 9
3.2 Quantitative Security Level of OMD-sha512 . . . 10

4 Specification of OMD **10**
4.1 The OMD Mode of Operation . . . 11
4.2 OMD-sha256: Primary Recommendation for Instantiating OMD . . . 12
4.3 OMD-sha512: Secondary Recommendation for Instantiating OMD . . . 15
4.4 Compression Functions of SHA-256 and SHA-512 . . . 16
4.4.1 Preliminaries . . . 16
4.4.2 The sha-256 Compression Function . . . 16
4.4.3 The sha-512 Compression Function . . . 18

5 Security Analysis **21**
5.1 Generalization of OMD based on Tweakable Random Functions . . . 21
5.2 Instantiating Tweakable RFs with PRFs . . . 25
5.2.1 Step 1. . . . 25
5.2.2 Step 2. . . . 25

6 Features **26**

7 Design Rationale **27**

8 Intellectual Property **28**

9 Consent **28**

A Changes from version 1.0 **31**

1 Introduction

An authenticated encryption (AE) scheme delivers on two complementary data security goals: confidentiality (privacy) and integrity (authenticity). Historically, these goals were achieved by combining separate cryptographic primitives, one to ensure confidentiality and another to guarantee integrity [7,8]. This generic composition paradigm is neither most efficient (for instance, it requires processing the input stream at least twice) nor most robust to implementation errors [11, 23]. To address these concerns, the notion of AE which simultaneously achieves confidentiality and integrity was put forward [7, 9, 16] and further developed [14, 18, 20–22] as a desirable primitive to be exposed by libraries and APIs to the end developer. Providing direct access to AE rather than requiring developers to make calls to several lower-level functions is seen as a step towards improving quality of security-critical code.

This document describes our proposal of a new authenticated cipher for consideration in the CAESAR competition. Our scheme, called Offset Merkle-Damgård (OMD), is a keyed compression function mode of operation for nonce-based AEAD. The syntax and security notions for nonce-based AEAD schemes were formalized by Rogaway in [18, 20]. To instantiate the OMD mode, we recommend two specific compression functions to be keyed and used in OMD, namely, the compression functions of the standard SHA-256 and SHA-512 hash functions. OMD parametrized with these two compression functions is called OMD-sha256 and OMD-sha512, respectively. The former is intended for 32-bit implementations and is our primary recommended algorithm, while the latter could be used specifically for 64-bit machines and is our secondary algorithm.

We believe that an AE scheme whose security is proved by a modular and easy to verify security reduction, only relying on some widely-verified standard assumption(s) on its underlying primitive(s), can get more confidence on its security compared to a scheme that demands strong and idealistic properties from its underlying primitive(s) or is not supported by a formal security proof. Provable security helps cryptanalysis efforts to be focused on analyzing the simpler underlying primitives rather than the whole scheme; hence, building confidence on the security of the scheme becomes easier if it uses well-analyzed and verified primitives.

Setting provable security in the standard model as one of our main design aims, OMD is designed as a scheme with its security goals achieved provably, based on the sole assumption that its underlying keyed compression function is a PRF; an assumption which is among the most well-known and widely-used assumption; for example, the security of the widely-employed standard HMAC algorithm is also based on this assumption [3]. From a theoretical point of view, this is an advantage for OMD compared to the recently proposed permutation-based AE schemes in the literature whose security proofs rely on the *ideal* permutation assumption.

Unlike the mainstream AE schemes which are blockcipher-based or permutation-based schemes, OMD is designed to be a compression function based scheme. The cryptographic community has spent more than two decades on public research and standardization activities on hash functions resulting to development of a rich source of secure and efficient compression functions. The recent announcement by Intel in July 2013 [2] about Intel SHA Extensions, supporting performance acceleration of the SHA family of functions (more precisely, SHA-1 and SHA-256), further encourages the decision to design a compression function based scheme. The SHA family of algorithms is heavily used in many of the most common cryptographic applications. For example, every secure web session initiation includes SHA-1, and the latest protocols involve SHA-256 as well. We believe that having a diverse set of AE schemes based on different primitives can be interesting from a practical viewpoint, providing the opportunity to choose among the AE algorithms based on what primitives have already been available and implemented and to reuse them.

OMD is patent free and suitable for widespread adoption. Our primary recommended scheme, OMD-sha256, uses the compression function of SHA-256 [1] and has features offering the following advantages over the AES-GCM scheme:

Higher quantitative security level. The proven security of OMD-sha256 falls off, as usual for birthday-type security bounds, in $\frac{\sigma^2}{2^{256}}$ where σ is the total number of calls to the compression function; while, for the same key size and tag size, the proven security of AES-GCM [15] falls off in about $\frac{\sigma'^2}{2^{128}}$ where σ' is the total number of calls to AES. That is, with the same key length and tag length, OMD-sha256 offers higher security level than that of AES-GCM.

More flexible key size. AES-GCM only supports three different key lengths, namely 128, 192 and 256 bits. OMD-sha256 can support any key length between 80 bits and 256 bits.

Simpler operations. OMD-sha256 only needs the compression function of SHA-256 plus the simple operations of bitwise XOR and bitwise AND of two binary strings and (left and right) shifting a binary string. In comparison, AES-GCM in addition to calling AES requires multiplication of two arbitrary elements in $\mathrm{GF}(2^{128})$. The field multiplication operation demand extra resources and is a complicated operation in contrast with the basic operations used in OMD-sha256. This is important, in particular, if one does not have access to Intel CPUs supporting the PCLMULQDQ instruction for implementing AES-GCM, e.g. on low-end devices.

Resistance against software-level timing attacks. Most AES software implementations risk leaking their keys through cache timing [10] unless they are implemented on machines with Intel® CPUs supporting the constant-time AES-NI and PCLMULQDQ instructions. In comparison, we note that the only operations in OMD-sha256 are: bitwise XOR, AND and OR of two binary strings (32-bit words in the compression function of SHA-256 and 256-bit words in the OMD iteration), fixed-distance (left and right) shift of a binary string (32-bit words in the compression function of SHA-256 and 256-bit words in the OMD iteration), and 32-bit addition (of words in the compression function of SHA-256). These operations have the virtue of taking constant time on typical CPUs in which case the implementations can avoid timing attacks.

Organization of this Document. The notations, definitions and concepts considered as preliminaries are presented in Section 2. Section 4 provides the specification of OMD as a mode of operation, and then the description of our two recommended instantiations: our primary recommended cipher, OMD-sha256, uses OMD with the compression function of the standard SHA-256 hash function; our secondary recommendation, OMD-sha512, uses OMD with the compression function of the standard SHA-512 hash function. We also provide the description of the compression functions of SHA-256 and SHA-512 for completeness. The security goals for OMD as an AEAD scheme are formally defined in Section 3. In Section 5, we provide the security analysis of OMD. In Section 6 we detail several interesting features of OMD. Section 7 provides an explanation of the main rationales behind the OMD design. Section 8 and Section 9, respectively, give the required information about the intellectual property and our consent to all decisions of the CAESAR selection committee.

2 Preliminaries

Notations. If $\mathcal{S}$ is a finite set, $x \xleftarrow{\$} S$ means that x is chosen from S uniformly at random. $X \leftarrow Y$ is used for denoting the assignment statement where the value of Y is assigned to X. The set of all binary strings of length n bits (for some positive integer n) is denoted as $\{0,1\}^n$, the set of all binary strings whose lengths are variable but upper-bounded by L is denoted by $\{0,1\}^{\leq L}$ and the set of all binary strings of arbitrary but finite length is denoted by $\{0,1\}^*$. For two strings X and Y we use $X||Y$ and XY analogously to denote the string obtained by concatenating Y to X. For an m-bit binary string $X = X_{m-1} \cdots X_0$ we denote the left-most bit by $\mathsf{msb}(X) = X_{m-1}$ and the right-most bit by $\mathsf{lsb}(X) = X_0$; let $X[i \cdots j] = X_i \cdots X_j$ denote

a substring of X, for $0 \le j \le i \le (m-1)$. Let $1^n 0^m$ denote concatenation of n ones by m zeros. For a non-negative integer i let $\langle i \rangle_m$ denote binary representation of i by an m-bit string.

For a binary string $X = X_{m-1} \cdots X_0$, let $X \ll n$ denote the left-shift operation, where the n left-most bits are discarded and the n vacated right bits are set to 0; that is, $X \ll n = X_{m-n-1} \cdots X_0 0^n$. We let $X \gg n$ denote the (unsigned) right-shift operation where the n right-most bits are discarded and the n vacated left bits are set to 0; i.e., $X \gg n = 0^n X_{m-1} \cdots X_n$. We let $X \gg_s n$ denote the *signed* right-shift operation where the n right-most bits are discarded and the n vacated left bits are filled with the left-most bit (which is considered as the sign bit); for example, $1001100 \gg_s 3 = 1111001$. If the left-most bit of X is 0 then we have $X \gg_s n = X \gg n$.

$\neg X$ means bitwise complement of X. For two binary strings X and Y, let $X \wedge Y$ and $X \vee Y$ denote, respectively, bitwise AND and bitwise OR of the strings.

The special symbol $\perp$ means that the value of a variable is undefined; we also overload this symbol and use it to signify an error. Let $|Z|$ denote the number of elements of Z if Z is a set, and the length of Z in bits if Z is a binary string. The empty string is denoted by ε and we let $|\varepsilon| = 0$. For $X \in \{0,1\}^*$ let $X[1]||X[2] \cdots ||X[m] \xleftarrow{b} X$ denote partitioning X into blocks $X[i]$ such that $|X[i]| = b$ for $1 \le i \le m-1$ and $|X[m]| \le b$; let $m = |X|_b$ denote length of X in b-bit blocks.

For two binary strings $X = X_{m-1} \cdots X_0$ and $Y = Y_{n-1} \cdots Y_0$, the notation $X \oplus Y$ denotes bitwise xor of $X_{m-1} \cdots X_{m-1-\ell}$ and $Y_{n-1} \cdots Y_{n-1-\ell}$ where $\ell = min\{m-1, n-1\}$. That is, $X \oplus Y$ is a binary string whose length is equal to the length of the shorter operand and is obtained by xoring the shorter operand with an equal length left-most substring of the longer operand consisting of its left-most bits. Clearly, if X and Y have the same length then $X \oplus Y$ simply means their usual bitwise xor. For any string X, define $X \oplus \varepsilon = \varepsilon \oplus X = \varepsilon$.

The Field with 2^n Points. Let $(GF(2^n), \oplus, .)$ denote the Galois Field with 2^n points. When considering a point α in $GF(2^n)$ it can be represented in any of the following equivalent ways: (1) as an integer between 0 and 2^n, (2) as a binary string $\alpha_{n-1} \cdots \alpha_0 \in \{0,1\}^n$, or (3) as a formal polynomial $\alpha(X) = \alpha_{n-1}X^{n-1} + \cdots + \alpha_1 X + \alpha_0$ with binary coefficients. For example, in $GF(2^{256})$: the string $0^{254}10$, the number 2 and the polynomial X are different representations of the same field element; the string $0^{254}11$, the number 3 and the polynomial $X+1$ represent the same field element, and so forth.

The addition "$\oplus$" and multiplication "." of two elements in $GF(2^n)$ are defined as follows. The addition of two elements $\alpha, \beta \in GF(2^n)$ simply means the element obtained by bitwise xoring their represenations as binary strings. For example, $2 \oplus 1 = 0^{n-2}10 \oplus 0^{n-2}01 = 0^{n-2}11 = 3$, $2 \oplus 3 = 1$, $1 \oplus 1 = 0$, and so forth. (Note that the addition operation in $GF(2^n)$ is *different* from the addition of integers module 2^n.) To multiply two elements, first choose and fix an irreducible polynomial $P_n(X)$ of degree n over GF(2); for example, choose the lexicographically first polynomial among the irreducible polynomials of degree n over GF(2) with a minimum number of nonzero coefficients. For example, for $n = 256$ we use $P_{256}(X) = X^{256} + X^{10} + X^5 + X^2 + 1$, for $n = 512$ we use $P_{512}(X) = X^{512} + X^8 + X^5 + X^2 + 1$.

To multiply two elements α and β in $GF(2^n)$ denoted by $\alpha \cdot \beta$ consider them as polynomials $\alpha(X) = \alpha_{n-1}X^{n-1} + \cdots + \alpha_1 X + \alpha_0$ and $\beta(X) = \beta_{n-1}X^{n-1} + \cdots + \beta_1 X + \beta_0$, form their product in GF(2) to get $\gamma(X)$ and take the remainder of dividing $\gamma(X)$ by the irreducible polynomial $P_n(X)$.

It is easy to multiply an arbitrary field element α by the element 2 (i.e. X). We describe this for $GF(2^{256})$and $GF(2^{512})$. Let $\alpha(X) = \alpha_{n-1}X^{n-1} + \cdots + \alpha_1 X + \alpha_0$ then multiplying by X we get $\alpha_n X^n + \alpha_{n-1}X^{n-1} \cdots + \alpha_1 X + \alpha_0 X$; so if $\mathsf{msb}(\alpha) = 0$ then $2.\alpha = X.\alpha = \alpha \ll 1$. If $\mathsf{msb}(\alpha) = 1$ then we need to reduce the result by module $P_n(X)$, i.e. we have to add X^n to $\alpha \ll 1$. For $n = 256$ using $P_{256}(X) = X^{256} + X^{10} + X^5 + X^2 + 1$, we have $X^{256} = X^{10} + X^5 + X^2 + 1 = 0^{245}10000100101$, so adding X^{256} means xoring with $0^{245}10000100101$. For $n = 512$ using $P_{512}(X) = X^{512} + X^8 + X^5 + X^2 + 1$, we have $X^{512} = X^8 + X^5 + X^2 + 1 = 0^{503}100100101$, so adding X^{512} means xoring with $0^{503}100100101$. In summary,

for $GF(2^{256})$

$$2.\alpha = \begin{cases} \alpha \ll 1 & \text{if } \mathsf{msb}(\alpha) = 0 \\ (\alpha \ll 1) \oplus 0^{245}10000100101 & \text{if } \mathsf{msb}(\alpha) = 1 \end{cases} \tag{1}$$

$$= (\alpha \ll 1) \oplus ((\alpha \gg_s 255) \wedge 0^{245}10000100101) \tag{2}$$

and for $GF(2^{512})$

$$2.\alpha = \begin{cases} \alpha \ll 1 & \text{if } \mathsf{msb}(\alpha) = 0 \\ (\alpha \ll 1) \oplus 0^{503}100100101 & \text{if } \mathsf{msb}(\alpha) = 1 \end{cases} \tag{3}$$

$$= (\alpha \ll 1) \oplus ((\alpha \gg_s 511) \wedge 0^{503}100100101) \tag{4}$$

We note that the results computed in (1) and (2) are the same but an implementation using (2) will not be susceptible to the timing attacks unlike one which uses (1). Similarly, an implementation using (4) is aimed for timing attack resistance.

Syntax of Keyed and Keyless Compression Functions. We denote a keyed compression function by $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$, where m and n are two positive integers, and the keyspace $\mathcal{K}$ is a non-empty set of strings. We write $F_K(H, M) = F(K; H, M)$ for every $K \in \mathcal{K}$, $H \in \{0,1\}^n$ and $M \in \{0,1\}^m$. We can alternatively think of F_K as a single argument function whose domain is $\{0,1\}^{n+m}$ and write $F_K(H||M) = F_K(H, M)$. If $|\mathcal{K}| = 1$ we assume that $\mathcal{K} = \{\varepsilon\}$, i.e. it only consists of the empty string, and in this case we call F a keyless compression function. Time_F denotes the time complexity of computing $F_K(X)$ for any $K \in \mathcal{K}$ and $X \in \{0,1\}^{n+m}$, plus the time complexity for sampling from $\mathcal{K}$.

Given a keyless compression function $F' : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$ (e.g. `sha-256` : $\{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$) we convert it to a keyed compression function F by borrowing k bits of its b-bit input block; i.e. we define $F_K(H, M) = F'(H, K||M)$.

Concrete Security Conventions. As usual in the concrete-security definitions, we use the resource parametrized function $\mathbf{Adv}_{\Pi}^{\mathsf{xxx}}(\mathbf{r})$ to denote the maximal value of the adversarial advantage (*i.e.* $\mathbf{Adv}_{\Pi}^{\mathsf{xxx}}(\mathbf{r}) = max_{\boldsymbol{A}} \{\mathbf{Adv}_{\Pi}^{\mathsf{xxx}}(\boldsymbol{A})\}$) over all adversaries $\boldsymbol{A}$, against the xxx property of a primitive or scheme Π, that use resources bounded by $\mathbf{r}$. The resource parameter $\mathbf{r}$, depending on the notion, may include time complexity (t), length of queries and number of queries that an adversary makes to its oracles. If a resource parameter is irrelevant in the context then we omit it; e.g. for information-theoretic security bounds the time complexity t is omitted.

Let $\boldsymbol{A}$ be an adversary that returns a binary value; by $\boldsymbol{A}^{f(.)}(X) \Rightarrow 1$ we refer to the event that $\boldsymbol{A}$ on input X and access to an oracle function $f(.)$ returns 1. By time complexity of an algorithm we mean the running time, relative to some fixed model of computation plus the size of the description of the algorithm using some fixed encoding method.

Pseudorandom Functions (PRFs) and Tweakable PRFs. Let $\text{Func}(m, n) = \{f : \{0,1\}^m \to \{0,1\}^n\}$ be the set of all functions from m-bit strings to n-bit strings. A random function (RF) R with m-bit input and n-bit output is a function selected uniformly at random from $\text{Func}(m, n)$. We denote this by $R \xleftarrow{\$} \text{Func}(m, n)$.

Let $\text{Func}^{\mathcal{T}}(m, n)$ be the set of all functions $\left\{\widetilde{f} : \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n\right\}$, where $\mathcal{T}$ is a set of tweaks. A tweakable RF with the tweak space $\mathcal{T}$, m-bit input and n-bit output is a map $\widetilde{R} : \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n$ selected uniformly at random from $\text{Func}^{\mathcal{T}}(m, n)$; i.e. $\widetilde{R} \xleftarrow{\$} \text{Func}^{\mathcal{T}}(m, n)$. Clearly, if $\mathcal{T} = \{0,1\}^t$ then $|\text{Func}^{\mathcal{T}}(m, n)| = |\text{Func}(m+t, n)|$, and hence, $\widetilde{R}$ can be instantiated using a random function R with $(m+t)$-bit input and n-bit output. We use $\widetilde{R}^{\langle T \rangle}(.)$ and $\widetilde{R}(T, .)$ interchangeably, for every $T \in \mathcal{T}$. Notice that each tweak T names a random function $\widetilde{R}^{\langle T \rangle} : \{0,1\}^m \to \{0,1\}^n$ and distinct tweaks name distinct (independent)

Let $F : \mathcal{K} \times \{0,1\}^m \to \{0,1\}^n$ be a keyed function and let $\widetilde{F} : \mathcal{K} \times \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n$ be a keyed and tweakable function, where the key space $\mathcal{K}$ is some nonempty set. Let $F_K(.) = F(K,.)$ and $\widetilde{F}_K^{\langle T \rangle}(.) = \widetilde{F}(K,T,.)$. Let $\boldsymbol{A}$ be an adversary. Then:

$$\begin{aligned}
\mathbf{Adv}_F^{\mathrm{prf}}(\boldsymbol{A}) &= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \boldsymbol{A}^{F_K(.)} \Rightarrow 1\right] - \Pr\left[R \xleftarrow{\$} \mathrm{Func}(m,n) : \boldsymbol{A}^{R(.)} \Rightarrow 1\right] \\
\mathbf{Adv}_{\widetilde{F}}^{\widetilde{\mathrm{prf}}}(\boldsymbol{A}) &= \Pr\left[K \xleftarrow{\$} \mathcal{K} : \boldsymbol{A}^{\widetilde{F}_K^{\langle . \rangle}(.)} \Rightarrow 1\right] - \Pr\left[\widetilde{R} \xleftarrow{\$} \mathrm{Func}^{\mathcal{T}}(m,n) : \boldsymbol{A}^{\widetilde{R}^{\langle . \rangle}(.)} \Rightarrow 1\right]
\end{aligned}$$

The resource parametrized advantage functions are defined accordingly, considering that the adversarial resources of interest here are the time complexity (t) of the adversary and the total number of queries (q) asked by the adversary (note that we just consider fixed-input-length functions, so the lengths of queries are fixed and known). We say that F is $(t,q;\epsilon)$-PRF if $\mathbf{Adv}_F^{\mathrm{prf}}(t,q) \leq \epsilon$. We say that $\widetilde{F}$ is $(t,q;\epsilon)$-tweakable PRF if $\mathbf{Adv}_{\widetilde{F}}^{\widetilde{\mathrm{prf}}}(t,q) \leq \epsilon$.

3 Security Goals

OMD is a nonce-based AEAD. Therefore, we aim to achieve the security notions for AEAD schemes as formalized in [18].

Nonce Respecting Adversaries. Let $\boldsymbol{A}$ be an adversary. We say that $\boldsymbol{A}$ is nonce-respecting if it never repeats a nonce in its *encryption* queries. That is, if $\boldsymbol{A}$ queries the encryption oracle $\mathcal{E}_K(\cdot,\cdot,\cdot)$ on $(N_1, A_1, M_1) \cdots (N_q, A_q, M_q)$ then $N_1, \cdots, N_q$ must be distinct.

In the following, we define the conventional security properties of an AEAD; namely, the privacy notion ("confidentiality for the plaintext") and the authenticity notion ("integrity for the nonce, associated data, and plaintext").

Privacy of AEAD Schemes. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a nonce-based AEAD scheme. Let $\boldsymbol{A}$ be a nonce-respecting adversary. $\boldsymbol{A}$ is provided with an oracle which can be either a real encryption oracle $\mathcal{E}_K(\cdot,\cdot,\cdot)$ such that on input (N, A, M) returns $\mathbb{C} = \mathcal{E}_K(N, A, M)$, or a fake encryption oracle $\$(\cdot,\cdot,\cdot)$ which on any input (N, A, M) returns $|\mathbb{C}|$ fresh random bits. The advantage of $\boldsymbol{A}$ in mounting a chosen plaintext attack (CPA) against the privacy property of Π is measured as follows:

$$\mathbf{Adv}_{\Pi}^{\mathrm{priv}}(\boldsymbol{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \boldsymbol{A}^{\mathcal{E}_K(\cdot,\cdot,\cdot)} \Rightarrow 1] - \Pr[\boldsymbol{A}^{\$(\cdot,\cdot,\cdot)} \Rightarrow 1].$$

This privacy notion, also called indistinguishability of ciphertext from random bits under CPA (IND\$-CPA), is defined originally in [21] and is a stronger variant of the classical IND-CPA notion [4, 7] for conventional symmetric-key encryption schemes.

Resource parameters for the CPA adversary. Let the CPA-adversary $\boldsymbol{A}$ make queries $(N_1, A_1, M_1) \cdots (N_{q_e}, A_{q_e}, M_{q_e})$. We define the resource parameters of $\boldsymbol{A}$ as $(t, q_e, \sigma_A, \sigma_M, L_{max})$ where t is the time complexity, q_e is the total number of encryption queries, $\sigma_A = \sum_{i=1}^{q} |A_i|$ is the total length of associated data in bits, $\sigma_M = \sum_{i=1}^{q} |M_i|$ is the total length of messages in bits, and L_{max} is the maximum length of each query in bits.

We remind that absence of a resource parameter means that the parameter is irrelevant in the context and hence omitted.

Authenticity of AEAD Schemes. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a nonce-based AEAD scheme. Let $\boldsymbol{A}$ be a nonce-respecting adversary. We stress that nonce-respecting is only regarded for the encryption queries; that

is, $\boldsymbol{A}$ can repeat nonces during its decryption queries and it can also ask an encryption query with a nonce that was already used in a decryption query. Let $\mathcal{A}$ be provided with the encryption oracle $\mathcal{E}_K(\cdot,\cdot,\cdot)$ and the decryption oracle $\mathcal{D}_K(\cdot,\cdot,\cdot)$; that is, we consider adversaries that can mount chosen ciphertext attacks (CCA). We say that $\boldsymbol{A}$ forges if it makes a decryption query $(N, A, \mathbb{C})$ such that $\mathcal{D}_K(N, A, \mathbb{C}) \neq \perp$ and no previous encryption query $\mathcal{E}_K(N, A, M)$ returned $\mathbb{C}$.

$$\mathbf{Adv}^{\mathrm{auth}}_{\Pi}(\boldsymbol{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \boldsymbol{A}^{\mathcal{E}_K(\cdot,\cdot,\cdot),\, \mathcal{D}_K(\cdot,\cdot,\cdot)} \text{ forges}].$$

This authenticity notion, also called integrity of ciphertext (INT-CTXT) under CCA attacks, is defined originally in [7].

Resource parameters for the CCA adversary. Let the CCA-adversary $\boldsymbol{A}$ make encryption queries $(N_1, A_1, M_1) \cdots (N_{q_e}, A_{q_e}, M_{q_e})$ and decryption queries $(N'_1, A'_1, \mathbb{C}'_1) \cdots (N'_{q_v}, A'_{q_v}, \mathbb{C}'_{q_v})$. We define the resource parameters of $\boldsymbol{A}$ as $(t, q_e, q_v, \sigma_A, \sigma_M, \sigma_{A'}, \sigma_{\mathbb{C}'}, L_{max})$, where t is the time complexity, q_e and q_v are respectively the total number of encryption queries and decryption queries, L_{max} is the maximum length of each query in bits, $\sigma_A = \sum_{i=1}^{q_e} |A_i|$, $\sigma_M = \sum_{i=1}^{q_e} |M_i|$, $\sigma_{A'} = \sum_{i=1}^{q_v} |A'_i|$ and $\sigma_{\mathbb{C}'} = \sum_{i=1}^{q_v} (|\mathbb{C}'_i| - \tau)$.

We remind that absence of a resource parameter means that the parameter is irrelevant in the context and hence omitted.

Remark 1 The use of the aforementioned privacy (IND\$-CPA) and authenticity (INT-CTXT) goals to define security of AE schemes dates back to [7] where it was shown that if an AE scheme satisfies the combination of IND-CPA and INT-CTXT properties then it will also fulfill indistinguishability under the strongest form of chosen-ciphertext attack (IND-CCA) which, in turn, is equivalent to non-malleability under chosen-ciphertext attack (NM-CCA).

Remark 2 The nonce-respecting assumption on the adversary is justified as follows. The nonce would typically be a counter (message number) maintained by the sender who encrypts the messages. In practice, an implementation must make sure that no nonce gets repeated within a session (i.e., the lifetime of the current encryption key). As the nonce N is needed both to encrypt and to decrypt; it would be typically communicated *in clear* between the sender and the receiver. Note that nonce-respecting is only assumed with respect to the encryption queries, reflecting the fact that the sender who encrypts a message is the party that is responsible for providing fresh nonces and the receiver may be stateless.

Remark 3 OMD v1.0 requires the nonce-respecting condition: it does not provide security if the nonce is repeated.

3.1 Quantitative Security Level of OMD-sha256

Using the concrete security bounds in Section 5 and letting $n = 256$ (the hash size for sha-256) one can calculate the quantitative security (privacy and authenticity) levels of OMD-sha256 for any set of fixed values for the adversarial resource parameters. For this purpose, we make the assumption that the function $F_K(H, M) = \text{sha-256}(H, K||0^{256-k}||M)$ is a PRF providing a k-bit security; as (to the best of our knowledge) there is no known attack with complexity less than 2^k against it. We note that having only a single (input, output) pair for F_K one can mount an *offline* exhaustive search attack with time complexity 2^k.

For the privacy property of OMD-sha256 (i.e. "confidentiality for the plaintext") the security bound falls off in $\frac{3\sigma_e^2}{2^{256}}$; that is, if the adversary has *online* data complexity about $\sigma_e = 2^{127}$, where σ_e denotes the total number of blocks in all inputs for encryption and decryption as defined in Section 3. According to the CAESAR call for submissions, algorithms should provide the intended "number of bits of security". We note that, giving a single measure for the bit security level of OMD-sha256 is a bit tricky as the terms determining

the security bound and the resources are different in nature (e.g. we have both offline complexity and online complexity); nevertheless, one can roughly consider $min\{k, 127\}$ as the bit security.

For the authenticity property of OMD-sha256 (i.e. "integrity for the public message number, the associated data and the plaintext") the security bound falls off in $\frac{3\sigma^2}{2^{256}} + \frac{q_v \ell_{max}}{2^{256}} + \frac{q_v}{2^\tau}$; that is, if the adversary has *online* data complexity about $\sigma_e = 2^{127}$, or $q_v \ell_{max} = 2^{256}$, or $q_v = 2^\tau$ (we refer to Section 3 for definitions of the resource parameters). As a single measure for the bit security of OMD-sha256 for the authenticity goal, one can roughly consider $min\{k, 127, \tau\}$.

Remark 4 We note that a single measure for the "bit security level" should be interpreted carefully regarding the different online/offline nature of the resources used for complexity measures. For example, just based on our bit security levels for OMD-sha256 one may think that a key length (k) larger than 127 bits or larger than the tag length (τ) is not useful, but this is not true because, for example, while the role of τ is to prevent online attacks, a large k can help prevent (mainly) offline key recovery attacks (that may only use one online query).

3.2 Quantitative Security Level of OMD-sha512

Using the concrete security bounds in Section 5 and letting $n = 512$ (the hash size for sha-512) one can calculate the quantitative security (privacy and authenticity) levels of OMD-sha512 for any set of fixed values for the adversarial resource parameters. For this purpose, we make the assumption that the function $F_K(H, M) = \text{sha-512}(H, K||0^{512-k}||M)$ is a PRF providing a k-bit security; as (to the best of our knowledge) there is no known attack with complexity less than 2^k against it. We note that having only a single (input, output) pair for F_K one can mount an *offline* exhaustive search attack with time complexity 2^k.

For the privacy property of OMD-sha512 (i.e. "confidentiality for the plaintext") the security bound falls off in $\frac{3\sigma_e^2}{2^{512}}$; that is, if the adversary has *online* data complexity about $\sigma_e = 2^{255}$, where σ_e denotes the total number of blocks in all inputs for encryption and decryption as defined in Section 3. According to the CAESAR call for submissions, algorithms should provide the intended "number of bits of security". We note that, giving a single measure for the bit security level of OMD-sha512 is a bit tricky as the terms determining the security bound and the resources are different in nature (e.g. we have both offline complexity and online complexity); nevertheless, one can roughly consider $min\{k, 255\}$ as the bit security.

For the authenticity property of OMD-sha512 (i.e. "integrity for the public message number, the associated data and the plaintext") the security bound falls off in $\frac{3\sigma^2}{2^{512}} + \frac{q_v \ell_{max}}{2^{512}} + \frac{q_v}{2^\tau}$; that is, if the adversary has *online* data complexity about $\sigma_e = 2^{255}$, or $q_v \ell_{max} = 2^{512}$, or $q_v = 2^\tau$ (we refer to Section 3 for definitions of the resource parameters). As a single measure for the bit security of OMD-sha512 for the authenticity goal, one can roughly consider $min\{k, 255, \tau\}$.

Remark 5 We note that a single measure for the "bit security level" should be interpreted carefully regarding the different online/offline nature of the resources used for complexity measures. For example, just based on our bit security levels for OMD-sha512 one may think that a key length (k) larger than 255 bits or larger than the tag length (τ) is not necessary, but this is not true because, for example, while the role of τ is to prevent online attacks, a large k can help prevent (mainly) offline key recovery attacks (that may only use one online query).

4 Specification of OMD

OMD is a compression function mode of operation for nonce-based authenticated encryption with associated data (AEAD). We use the syntax of AEAD schemes following [18].

Syntax of an AEAD Scheme. A nonce-based authenticated encryption with associated data, AEAD for short, is a symmetric key scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key space $\mathcal{K}$ is some non-empty finite set. The encryption algorithm $\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \to \mathcal{C} \cup \{\perp\}$ takes four arguments, a secret key $K \in \mathcal{K}$, a nonce $N \in \mathcal{N}$, an associated data (a.k.a. header data) $A \in \mathcal{A}$ and a message $M \in \mathcal{M}$, and returns either a ciphertext $\mathbb{C} \in \mathcal{C}$ or a special symbol $\perp$ indicating an error. The decryption algorithm $\mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C} \to \mathcal{M} \cup \{\perp\}$ takes four arguments $(K, N, A, \mathbb{C})$ and either outputs a message $M \in \mathcal{M}$ or an error indicator $\perp$.

For correctness of the scheme, it is required that $\mathcal{D}(K, N, A, \mathbb{C}) = M$ for any $\mathbb{C}$ such that $\mathbb{C} = \mathcal{E}(K, N, A, M)$. It is also assumed that if algorithms $\mathcal{E}$ and $\mathcal{D}$ receive parameter not belonging to their specified domain of arguments they will output $\perp$. We write $\mathcal{E}_K(N, A, M) = \mathcal{E}(K, N, A, M)$ and similarly $\mathcal{D}_K(N, A, \mathbb{C}) = \mathcal{D}(K, N, A, \mathbb{C})$.

We assume that the message and associated data can be any binary string of arbitrary but finite length; i.e. $\mathcal{M} = \{0,1\}^*$ and $\mathcal{A} = \{0,1\}^*$, but the key and nonce are some fixed-length binary strings, i.e. $\mathcal{N} = \{0,1\}^{|N|}$ and $\mathcal{K} = \{0,1\}^k$, where the positive integers $|N|$ and k are respectively the nonce length and the key length of the scheme in bits. We assume that $|\mathcal{E}_K(N, A, M)| = |M| + \tau$ for some positive fixed constant τ; that is, we will have $\mathbb{C} = C||\mathsf{Tag}$ where $|C| = |M|$ and $|\mathsf{Tag}| = \tau$. We call C the core ciphertext and Tag the tag.

Remark 6 According to the CAESAR call for proposals "An authenticated cipher is a function with five byte-string inputs and one byte-string output. The five inputs are a variable-length plaintext, variable-length associated data, a fixed-length secret message number, a fixed-length public message number, and a fixed-length key. The output is a variable-length ciphertext." OMD considers the "public message number" as the nonce and does not support a secret message number.

4.1 The OMD Mode of Operation

To use OMD one must specify a keyed compression function $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ and a tag length $\tau \leq n$; where the key space $\mathcal{K} = \{0,1\}^k$ and $m \leq n$ where the case $m = n$ is the optimal choice from efficiency viewpoint. At first glance, requiring $m \leq n$ may look a bit odd as usually a compression function has a larger input block length than its output (hash) length, so we first explain this restriction based on the following two observations:

- It will be clear from the description of OMD in the sequel that at each call to the compression function only n random bits (namely, the output bits of the compression function) are available for encrypting an m-bit message block, hence we must have $m \leq n$. The optimal case is when $m = n$, so no random bits are wasted. We notice that this limitation applies to any compression function based AE, therefore a compression function based AE scheme (like OMD) will be usually less efficient than a blockcipher based AE (like OCB) "unless" one uses a dedicated compression function which is more efficient than the blockcipher.

- In practice, the compression function of standard hash functions (e.g. SHA-1 or the SHA-2 family) are keyless, i.e. do not have a dedicated key input, therefore one will need to use k bits of their b-bit message block to get a keyed function. So, there will be no efficiency waste in each call to the compression function if $m = n$ and $b = n + k$; for example, when the key length is 256 bits and the compression function of SHA-256 is used.

We let OMD-F denote the OCB mode of operation using a keyed compression function $F_K : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ with $m \leq n$ and an unspecified tag length. We let OMD$[F, \tau]$ denote the OMD mode of operation using keyed compression function F_K and tag length τ. The encryption algorithm of OMD$[F, \tau]$ inputs four arguments (secret key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^{|N|}$, associated data $A \in \{0,1\}^*$, message

$M \in \{0,1\}^*$) and outputs $\mathbb{C} = C||\mathsf{Tag} \in \{0,1\}^{|M|+\tau}$. The decryption algorithm of OMD$[F,\tau]$ inputs four arguments (secret key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^{|N|}$, associated data $A \in \{0,1\}^*$, ciphertext $C||\mathsf{Tag} \in \{0,1\}^*$) and either outputs the whole $M \in \{0,1\}^{|C|-\tau}$ at once or an error message ($\perp$). Note that we have either $C = C_1 \cdots C_\ell$ or $C = C_1 \cdots C_{\ell-1} C_*$ depending on whether the message length in bits is a multiple of the block length m or not, respectively.

Figure 1 depicts the construction of the encryption algorithm of OMD$[F,\tau]$. The construction of the decryption algorithm is straightforward and almost the same as the encryption algorithm except a tag comparison (verification) at the end of the decryption process. Figure 2 describes the encryption and decryption algorithms of OMD$[F,\tau]$. We remind that for two binary strings $X = X_{m-1} \cdots X_0$ and $Y = Y_{n-1} \cdots Y_0$, the notation $X \oplus Y$ denotes bitwise xor of $X_{m-1} \cdots X_{m-1-\ell}$ and $Y_{n-1} \cdots Y_{n-1-\ell}$ where $\ell = min\{m-1, n-1\}$.

Computing the masking values. As seen from the description of OMD in Figure 1, before each call to the underlying keyed compression function we xor a masking value denoted as $\Delta_{N,i,j}$ (the top and middle parts of Figure 1) and $\bar{\Delta}_{i,j}$ (the bottom part of Figure 1). In the following, we describe how these masks are generated. We note that there are both security and efficient related criteria to be satisfied by the method to compute the masking values. We only explain the efficiency criterion for computing the masks here; the security related properties will be made clear in Section 5. By an efficient masking scheme, we mean a scheme in which the mask value needed for processing a block can be efficiently computed from the mask value used for processing the previous block.

There are different ways to compute the masking values to satisfy both the security and efficiency criteria; for example, we refer to [12, 17, 19]. We use the method proposed in [17].

In the following, all multiplications are in $GF(2^n)$, $\mathtt{ntz}(i)$ denotes the number of trailing zeros (i.e. the number of rightmost bits that are zero) in the binary representation of a positive integer i.

Initialization. $\Delta_{N,0,0} = F_K(N||10^{n-1-|N|}, 0^m)$; $\bar{\Delta}_{0,0} = 0^n$; $L_* = F_K(0^n, \langle\tau\rangle_m)$; $L[0] = 4.L_*$, and $L[i] = 2.L[i-1]$ for $i \geq 1$. We note that the values $L[i]$ can be preprocessed and stored (for a fast implementation) in a table for $0 \leq i \leq \lceil \log_2(\ell_{max}) \rceil$, where ℓ_{max} is the the bound on the maximum number of m-bit blocks in any message that can be encrypted or decrypted. Alternatively, (if there is a memory restriction) they can be computed on-the-fly for $i \geq 1$. It is also possible to precompute and store some values and then compute the others as needed on-the-fly.

Masking sequence for processing the message. For $i \geq 1$: $\Delta_{N,i,0} = \Delta_{N,i-1,0} \oplus L[\mathtt{ntz}(i)]$; $\Delta_{N,i,1} = \Delta_{N,i,0} \oplus 2.L_*$; and $\Delta_{N,i,2} = \Delta_{N,i,0} \oplus 3.L_*$.

Masking sequence for processing the associate data. $\bar{\Delta}_{i,0} = \bar{\Delta}_{i-1,0} \oplus L[\mathtt{ntz}(i)]$ for $i \geq 1$; and $\bar{\Delta}_{i,1} = \bar{\Delta}_{i,0} \oplus L_*$ for $i \geq 0$.

4.2 OMD-sha256: Primary Recommendation for Instantiating OMD

Our primary recommendation to instantiate OMD is called OMD-sha256 and uses the underlying compression function of SHA-256 [1]. This is intended to be the appropriate choice for implementations on 32-bit machines. The compression function of SHA-256 is a map sha-256 : $\{0,1\}^{256} \times \{0,1\}^{512} \rightarrow \{0,1\}^{256}$. On input a 256-bit chaining block X and a 512-bit message block Y, it outputs a 256-bit digest Z, i.e. let $Z = \text{sha-256}(X, Y)$. The description of sha-256 is provided in subsection 4.4.

To use OMD with sha-256, we use the first 256-bit argument X for chaining values as usual. In our notation (see Figure 1) this means that $n = 256$. We use the 512-bit argument Y (the message block

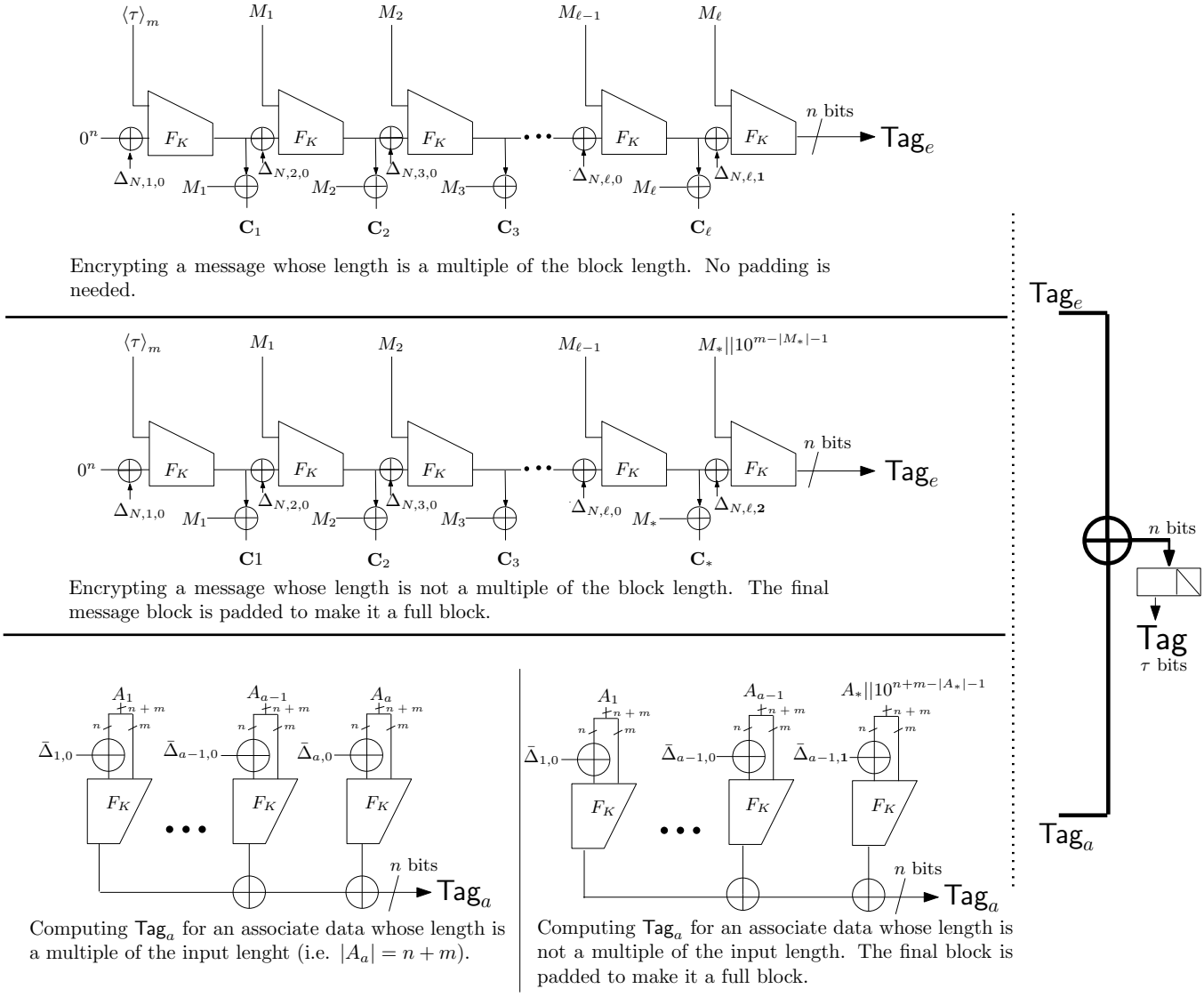

Figure 1: The encryption process of OMD$[F, \tau]$ using a keyed compression function $F_K : (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ with $m \leq n$. (**TOP**) The encryption process when the message length is a multiple of the block length m and no padding is required. (**Middle**) The encryption process when the message length is not a multiple of the block length and the final block M_* is padded to make a full block $M_*||10^{m-|M_*|-1}$. (**Bottom, Left**) Computing the intermediate value T_a when the bit length of the associated data is a multiple of the **input** length $n+m$. (**Bottom, Right**) Computing T_a when the bit length of the associated data is not a multiple of $n+m$ and the final block is padded to make a full block $A_*||10^{n+m-|A_*|-1}$ is needed. The output ciphertext is $C||\mathsf{Tag}$. For operation $\oplus$ see our convention in Section 2. Five types of key-dependent masking values (corresponding to five mutually exclusive tweak sets) are used; these are denoted by $\Delta_{N,i,0}, \Delta_{N,i,1}, \Delta_{N,i,2}, \bar{\Delta}_{i,0}$ and $\bar{\Delta}_{j,1}$, for $i \geq 1$ and $j \geq 0$, where N is the nonce. Note that the masks used in computing T_a do not depend on the nonce.

Algorithm $\textsc{Initialize}(K)$
- $L_* \leftarrow F_K(0^n, \langle\tau\rangle_m)$
- $L[0] \leftarrow 4.L_*$ ▷ $2.(2.L_*)$, doubling in $GF(2^n)$
- **for** $i \leftarrow 1$ **to** $\lceil \log_2(\ell_{max}) \rceil$ **do**
  - $L[i] = 2.L[i-1]$ ▷ doubling in $GF(2^n)$
- **return**

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

Algorithm $\mathcal{E}_K(N, A, M)$
- **if** $|N| > n - 1$ **then**
  - **return** $\perp$
- $M_1 || M_2 \cdots M_{\ell-1} || M_\ell \xleftarrow{m} M$, where $|M_i| = m$ for $1 \le i \le \ell - 1$ and $|M_\ell| \le m$
- $\Delta \leftarrow F_K(N || 10^{n-1-|N|}, 0^m)$ ▷ 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**
  - $C_i \leftarrow H \oplus M_i$
  - $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(i+1)]$
  - $H \leftarrow F_K(H \oplus \Delta, M_i)$
- $C_\ell \leftarrow H \oplus M_\ell$
- **if** $|M_\ell| = m$ **then**
  - $\Delta \leftarrow \Delta \oplus 2.L_*$
  - $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell)$
- **else if** $|M_\ell| \neq 0$ **then**
  - $\Delta \leftarrow \Delta \oplus 3.L_*$
  - $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell || 10^{m-|M_\ell|-1})$
- **else**
  - $\mathsf{Tag}_e \leftarrow H$
- $\mathsf{Tag}_a \leftarrow \text{HASH}_K(A)$
- $\mathsf{Tag} \leftarrow (\mathsf{Tag}_e \oplus \mathsf{Tag}_a)[n-1 \cdots n-\tau]$
- $\mathbb{C} \leftarrow C_1 || C_2 || \cdots || C_\ell || \mathsf{Tag}$
- **return** $\mathbb{C}$

Algorithm $\mathcal{D}_K(N, A, \mathbb{C})$
- **if** $|N| > n - 1$ **or** $|\mathbb{C}| < \tau$ **then**
  - **return** $\perp$
- $C_1 || C_2 \cdots C_{\ell-1} || C_\ell || \mathsf{Tag} \xleftarrow{m} \mathbb{C}$, where $|C_i| = m$ for $1 \le i \le \ell - 1$, $|C_\ell| \le m$ 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**
  - $M_i \leftarrow H \oplus C_i$
  - $\Delta \leftarrow \Delta \oplus L[\mathtt{ntz}(i+1)]$
  - $H \leftarrow F_K(H \oplus \Delta, M_i)$
- $M_\ell \leftarrow H \oplus C_\ell$
- **if** $|C_\ell| = m$ **then**
  - $\Delta \leftarrow \Delta \oplus 2.L_*$
  - $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell)$
- **else if** $|C_\ell| \neq 0$ **then**
  - $\Delta \leftarrow \Delta \oplus 3.L_*$
  - $\mathsf{Tag}_e \leftarrow F_K(H \oplus \Delta, M_\ell || 10^{m-|M_\ell|-1})$
- **else**
  - $\mathsf{Tag}_e \leftarrow H$
- $\mathsf{Tag}_a \leftarrow \text{HASH}_K(A)$
- $\mathsf{Tag}' \leftarrow (\mathsf{Tag}_e \oplus \mathsf{Tag}_a)[n-1 \cdots n-\tau]$
- **if** $\mathsf{Tag}' = \mathsf{Tag}$ **then**
  - **return** $M \leftarrow M_1 || M_2 || \cdots || M_\ell$
- **else**
  - **return** $\perp$

Figure 2: Definition of $\text{OMD}[F, \tau]$. The function $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ is a keyed compression function with $\mathcal{K} = \{0,1\}^k$ and $m \le n$. The tag length is $\tau \in \{0, 1, \cdots, n\}$. Algorithms $\mathcal{E}$ and $\mathcal{D}$ can be called with arguments $K \in \mathcal{K}$, $N \in \{0,1\}^{\le n-1}$, and $A, M, \mathbb{C} \in \{0,1\}^*$. ℓ_{max} is the bound on the maximum number of blocks in any input to the encryption or decryption algorithms.

in sha-256) to input both a 256-bit message block and the key K which can be of any length $k \leq 256$ bits. If $k < 256$ then let the key be $K||0^{256-k}$. That is, we define the keyed compression function $F_K : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0,1\}^{256}$ needed in OMD as $F_K(H, M) = \text{sha-256}(H, K||0^{256-k}||M)$.

The parameters of OMD-sha256 are as follows:

- The message block length in bits is $m = 256$; i.e. $|M_i| = 256$. *If needed*, we pad the final block of the message with 10^* (i.e., a single 1 followed by the minimal number of 0's needed) to make its length exactly 256 bits.
- The key length in bits can be $80 \leq k \leq 256$; but $k < 128$ is not recommended. *If needed*, we pad the key K with 0^{256-k} to make its length exactly 256 bits.
- The nonce (public message number) length in bits can be $96 \leq |N| \leq 255$. We *always* pad the nonce with $10^{255-|N|}$ to make its length exactly 256 bits.
- The secret message number length in bits is 0; that is, our scheme does not support secret message numbers.
- The associated data block length in bits is $2n = 512$; i.e. $|A_i| = 512$. *If needed*, we pad the final block of the associated data with 10^* (i.e., a single 1 followed by the minimal number of 0's needed) to make its length exactly 512 bits.
- The tag length in bits can be $32 \leq \tau \leq 256$; but it must be noted that the selection of the tag length directly affects the achievable security level. We refer to Section 5 for the security bounds.

4.3 OMD-sha512: Secondary Recommendation for Instantiating OMD

Our secondary recommendation to instantiate OMD is called OMD-sha512 and uses the underlying compression function of SHA-512 [1]. This is intended to be the appropriate choice for implementations on 64-bit machines. The compression function of SHA-512 is a map $\text{sha-512} : \{0,1\}^{512} \times \{0,1\}^{1024} \to \{0,1\}^{512}$. On input a 512-bit chaining block X and a 1024-bit message block Y, it outputs a 512-bit digest Z, i.e. let $Z = \text{sha-512}(X, Y)$. The description of sha-512 is provided in subsection 4.4.

To use OMD with sha-512, we use the first 512-bit argument X for chaining values as usual. In our notation (see Figure 1) this means that $n = 512$. We use the 1024-bit argument Y (the message block in sha-512) to input both a 512-bit message block and the key K which can be of any length $k \leq 512$ bits. If $k < 512$ then let the key be $K||0^{512-k}$. That is, we define the keyed compression function $F_K : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$ needed in OMD as $F_K(H, M) = \text{sha-512}(H, K||0^{512-k}||M)$.

The parameters of OMD-sha512 are set as follows:

- The message block length in bits is $m = 512$; i.e. $|M_i| = 512$. *If needed*, we pad the final block of the message with 10^* (i.e., a single 1 followed by the minimal number of 0's needed) to make its length exactly 512 bits.
- The key length in bits can be $80 \leq k \leq 512$; but $k < 128$ is not recommended. *If needed*, we pad the key K with 0^{512-k} to make its length exactly 512 bits.
- The nonce (public message number) length in bits can be $96 \leq |N| \leq 511$. We *always* pad the nonce with $10^{511-|N|}$ to make its length exactly 512 bits.
- The secret message number length in bits is 0; that is, our scheme does not support secret message numbers.

- The associated data block length in bits is $2n = 1024$; i.e. $|A_i| = 1024$. *If needed*, we pad the final block of the associated data with 10^* (i.e., a single 1 followed by the minimal number of 0's needed) to make its length exactly 1024 bits.
- The tag length in bits can be $32 \leq \tau \leq 512$; but it must be noted that the selection of the tag length directly affects the achievable security level. We refer to Section 5 for the security bounds.

4.4 Compression Functions of SHA-256 and SHA-512

The CAESAR call for submissions has mentioned that "The cipher definition is required to be self-contained, including all information necessary to implement the cipher from scratch, except that the following functions are free to be used without being defined: AES-128 with 128-bit key, 128-bit input, and 128-bit output; AES-192 with 192-bit key, 128-bit input, and 128-bit output; AES-256 with 256-bit key, 128-bit input, and 128-bit output."

Therefore, in this section we include a description of the compression functions of the standard SHA-256 and SHA-512 hash functions from NIST FIPS PUB 180-4 [1]. We refer to the underlying compression functions of these standard hash functions as sha-256 and sha-512, respectively.

4.4.1 Preliminaries

In the following, by "word" we mean a group of either 32 bits (4 bytes) or 64 bits (8 bytes), depending on the compression function algorithm. Namely, in sha-256 each word is a 32-bit string and in sha-512 each word is a 64-bit string.

$\mathbf{ROTR^n(x)}$: The *rotate right* (circular right shift) operation, where x is a w-bit word and n an integer with $0 \leq n < w$, is defined by $\mathrm{ROTR}^n(x) = (x \gg n) \vee (x \ll w - n)$

$\mathbf{SHR^n(x)}$: The *right shift* operation, where x is a w-bit word and n an integer with $0 \leq n < w$, is defined by $\mathrm{SHR}^n(x) = (x \gg n)$.

The addition $x + y$ of two w-bit words x and y is defined as follows. The words x and y represent integers X and Y, where $0 \leq X < 2^w$ and $0 \leq Y < 2^w$. Compute $Z = (X + Y) \bmod 2^w$. Then $0 \leq Z < 2^w$. Convert the integer Z to a word z and define $z = x + y$.

4.4.2 The sha-256 Compression Function

sha-256 uses six logical functions, where each function operates on 32-bit words, which are represented as x, y, and $z$and outputs a 32-bit word as a result. These functions are defined as follows:

$$Ch : \{0,1\}^{32} \times \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{32}, \qquad Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj : \{0,1\}^{32} \times \{0,1\}^{32} \times \{0,1\}^{32} \to \{0,1\}^{32}, \quad Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0^{\{256\}} : \{0,1\}^{32} \to \{0,1\}^{32}, \quad \Sigma_0^{\{256\}}(x) = \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x)$$
$$\Sigma_1^{\{256\}} : \{0,1\}^{32} \to \{0,1\}^{32}, \quad \Sigma_1^{\{256\}}(x) = \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x)$$
$$\sigma_0^{\{256\}} : \{0,1\}^{32} \to \{0,1\}^{32}, \quad \sigma_0^{\{256\}}(x) = \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x)$$
$$\sigma_1^{\{256\}} : \{0,1\}^{32} \to \{0,1\}^{32}, \quad \sigma_1^{\{256\}}(x) = \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x)$$

During the process of compression, a sequence of 64 constant 32-bit words, $K_0^{\{256\}}, ..., K_{63}^{\{256\}}$ are used. These 32-bit words represent the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers. In hex, these constant words are (from left to right):

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

The Compression Process. The sha-256 compression function is defined as follows:

$$\text{sha-256} : \{0,1\}^{256} \times \{0,1\}^{512} \longrightarrow \{0,1\}^{256}, \qquad \text{sha-256}(H, M) = C$$

Let H be the 256-bit *hash input* (chaining input) and M be the 512-bit *message input.* These two inputs are represented respectively by an array of 8 32-bit words $H_0 \cdots H_7$ and an array of 16 32-bit words $M_0 \cdots M_{15}$. The 256-bit output value C is also represented as an array of 8 32-bit words $C_0 \cdots C_7$.

The compression function processes as below:

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

$$W_t = \begin{cases} M_t & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

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

$$\begin{aligned} a &= H_0 \\ b &= H_1 \\ c &= H_2 \\ d &= H_3 \\ e &= H_4 \\ f &= H_5 \\ g &= H_6 \\ h &= H_7 \end{aligned}$$

3. For $t = 0$ to 63, do:

   {

$$
\begin{aligned}
T_1 &= h + \Sigma_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t \\
T_2 &= \Sigma_0^{\{256\}}(a) + Maj(a,b,c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned}
$$

}

4. Compute the 256-bit output (hash) value $C = C_0 \cdots C_7$ as:

$$
\begin{aligned}
C_0 &= a + H_0 \\
C_1 &= b + H_1 \\
C_2 &= c + H_2 \\
C_3 &= d + H_3 \\
C_4 &= e + H_4 \\
C_5 &= f + H_5 \\
C_6 &= g + H_6 \\
C_7 &= h + H_7
\end{aligned}
$$

4.4.3 The sha-512 Compression Function

sha-512 uses six logical functions, where each function operates on 64-bit words, which are represented as x, y, and $z$and outputs a 64-bit word as a result. These functions are defined as follows:

$$
\begin{aligned}
Ch &: \{0,1\}^{64} \times \{0,1\}^{64} \times \{0,1\}^{64} \to \{0,1\}^{64}, \qquad Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \\
Maj &: \{0,1\}^{64} \times \{0,1\}^{64} \times \{0,1\}^{64} \to \{0,1\}^{64}, \quad Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_0^{\{512\}} &: \{0,1\}^{64} \to \{0,1\}^{64}, \quad \Sigma_0^{\{256\}}(x) = \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x) \\
\Sigma_1^{\{512\}} &: \{0,1\}^{64} \to \{0,1\}^{64}, \quad \Sigma_1^{\{256\}}(x) = \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x) \\
\sigma_0^{\{512\}} &: \{0,1\}^{64} \to \{0,1\}^{64}, \quad \sigma_0^{\{256\}}(x) = \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x) \\
\sigma_1^{\{512\}} &: \{0,1\}^{64} \to \{0,1\}^{64}, \quad \sigma_1^{\{256\}}(x) = \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x)
\end{aligned}
$$

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 (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

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

$$\text{sha-512} : \{0,1\}^{512} \times \{0,1\}^{1024} \longrightarrow \{0,1\}^{512}, \qquad \text{sha-512}(H, M) = C$$

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 $H_0 \cdots H_7$ and an array of 16 64-bit words $M_0 \cdots M_{15}$. The 512-bit output value C is also represented as an array of 8 64-bit words $C_0 \cdots C_7$.

The compression function processes as described below:

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

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

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

$$\begin{aligned} a &= H_0 \\ b &= H_1 \\ c &= H_2 \\ d &= H_3 \\ e &= H_4 \\ f &= H_5 \\ g &= H_6 \\ h &= H_7 \end{aligned}$$

3. For $t = 0$ to 79, do:

$$
\begin{aligned}
&\{\\
&T_1 &= & h + \Sigma_1^{\{512\}}(e) + Ch(e,f,g) + K_t^{\{512\}} + W_t\\
&T_2 &= & \Sigma_0^{\{512\}}(a) + Maj(a,b,c)\\
&h &= & g\\
&g &= & f\\
&f &= & e\\
&e &= & d + T_1\\
&d &= & c\\
&c &= & b\\
&b &= & a\\
&a &= & T_1 + T_2\\
&\}
\end{aligned}
$$

4. Computing the 512-bit output (hash) value $C = C_0 \cdots C_7$ as:

$$
\begin{aligned}
C_0 &= a + H_0\\
C_1 &= b + H_1\\
C_2 &= c + H_2\\
C_3 &= d + H_3\\
C_4 &= e + H_4\\
C_5 &= f + H_5\\
C_6 &= g + H_6\\
C_7 &= h + H_7
\end{aligned}
$$

5 Security Analysis

Theorem 1 provides the security bounds of OMD.

Theorem 1 *Fix $n \geq 1$ and $\tau \in \{0, 1, \cdots, n\}$. Let $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be a PRF, where the key space $\mathcal{K} = \{0,1\}^k$ for $k \geq 1$ and $1 \leq m \leq n$. Then*

$$\begin{aligned}
\mathbf{Adv}^{\text{priv}}_{\text{OMD}[F,\tau]}(t, q_e, \sigma_e, \ell_{max}) &\leq \mathbf{Adv}^{\text{prf}}_F(t', 2\sigma_e) + \frac{3\sigma_e^2}{2^n} \\
\mathbf{Adv}^{\text{auth}}_{\text{OMD}[F,\tau]}(t, q_e, q_v, \sigma, \ell_{max}) &\leq \mathbf{Adv}^{\text{prf}}_F(t', 2\sigma) + \frac{3\sigma^2}{2^n} + \frac{q_v \ell_{max}}{2^n} + \frac{q_v}{2^\tau}
\end{aligned}$$

where q_e and q_v are, respectively, the number of encryption and decryption queries, ℓ_{max} denotes the maximum number of m-bit blocks in an encryption or decryption query, $t' = t + cn\sigma$ for some constant c, and σ_e and σ are the total number of calls to the underlying compression function F in all queries asked by the CPA and CCA adversaries against the privacy and authenticity of the scheme, respectively.

The proof is obtained by combing Lemma 2 in subsection 5.1 with Lemma 3 and Lemma 4 in subsection 5.2.

Remark 7 Referring to subsection 3 for definitions of the resource parameters, it can be seen that: $\sigma_e = \lceil \sigma_M/m \rceil + \lceil \sigma_A/(n+m) \rceil + q_e + 2$; $\sigma = \lceil (\sigma_M + \sigma_{\mathbb{C}'})/m \rceil + \lceil (\sigma_A + \sigma_{A'})/(n+m) \rceil + q + 2$; and $\ell_{max} = \lceil L_{max}/m \rceil$.

5.1 Generalization of OMD based on Tweakable Random Functions

Figure 3 shows the $\mathbb{OMD}[\widetilde{R}, \tau]$ scheme which is a generalization of $OMD[F, \tau]$ using a tweakable random function $\widetilde{R} : \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$. The tweak space $\mathcal{T}$ consists of five mutually exclusive sets of tweaks; namely, $\mathcal{T} = \mathcal{N} \times \mathbb{N} \times \{0\} \cup \mathcal{N} \times \mathbb{N} \times \{1\} \cup \mathcal{N} \times \mathbb{N} \times \{2\} \cup \mathbb{N} \times \{0\} \cup \mathbb{N} \times \{1\}$, where $\mathcal{N} = \{0,1\}^{|N|}$ is the set of nonces and $\mathbb{N}$ is the set of positive integers.

Lemma 2 *Let $\mathbb{OMD}[\widetilde{R}, \tau]$ be the scheme shown in Figure 3. Then*

$$\begin{aligned}
\mathbf{Adv}^{\text{priv}}_{\mathbb{OMD}[\widetilde{R},\tau]}(q_e, \sigma_e, \ell_{max}) &= 0 \\
\mathbf{Adv}^{\text{auth}}_{\mathbb{OMD}[\widetilde{R},\tau]}(q_e, q_v, \sigma, \ell_{max}) &\leq \frac{q_v \ell_{max}}{2^n} + \frac{q_v}{2^\tau}
\end{aligned}$$

where q_e and q_v are, respectively, the number of encryption and decryption queries, ℓ_{max} denotes the maximum number of m-bit blocks in an encryption or decryption query, and σ_e and σ are the total number of calls to the underlying tweakable random function $\widetilde{R}$ in all queries asked by the CPA and CCA adversaries against the privacy and authenticity of the scheme, respectively.

The proof of the privacy bound is straightforward. Let $\boldsymbol{A}$ be a CPA adversary that asks (encryption) queries $(N^1, A^1, M^1) \cdots (N^{q_e}, A^{q_e}, M^{q_e})$ where all N^x values (for $1 \leq x \leq q_e$) are distinct due to the nonce-respecting assumption on the adversary $\boldsymbol{A}$. Referring to Figure 3, this means that we are applying independent random functions $\widetilde{R}^{N_x,i,j}$ each to a single point, hence the images that the adversary sees (i.e. $\mathbb{C}^x$ for $1 \leq x \leq q_e$) are fresh uniformly random values.

The authenticity bound can be shown by a straightforward but lengthy case analysis. First we consider the single verification case where the adversary only makes one decryption (verification) query and then we

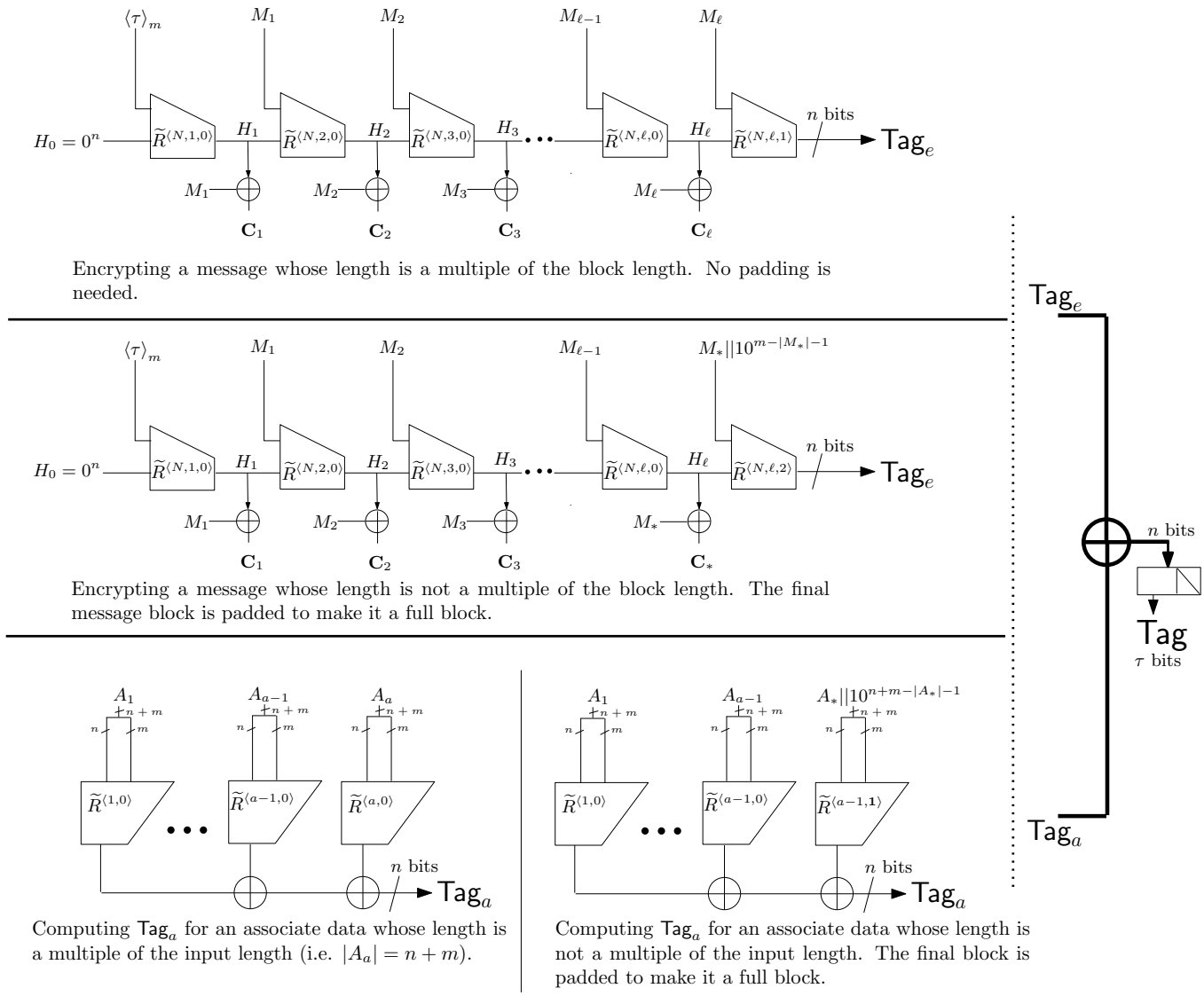

Figure 3: The $\mathbb{OMD}[\widetilde{R}, \tau]$ scheme using a tweakable random function $\widetilde{R} : \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ (i.e. $\widetilde{R} \xleftarrow{\$} \mathrm{Func}^{\mathcal{T}}(n+m, n)$). The tweak space $\mathcal{T}$ consists of five mutually exclusive sets of tweaks; namely, $\mathcal{T} = \mathcal{N} \times \mathbb{N} \times \{0\} \cup \mathcal{N} \times \mathbb{N} \times \{1\} \cup \mathcal{N} \times \mathbb{N} \times \{2\} \cup \mathbb{N} \times \{0\} \cup \mathbb{N} \times \{1\}$, where $\mathcal{N} = \{0,1\}^{|N|}$ is the set of nonces, $\mathbb{N}$ is the set of positive integers.

will use the generic result of Bellare et al. [5] to get a bound against adversaries that make multiple (say q_v) verification queries. Let $\boldsymbol{A}$ be a CCA adversary making encryption queries $(N^1, A^1, M^1) \cdots (N^{q_e}, A^{q_e}, M^{q_e})$. Let $M^i = M^i_1 \cdots M^i_{\ell_i}$ or $M^i = M^i_1 \cdots M^i_{\ell_i - 1} M^i_*$ be the message queries and $A^i = A^i_1 \cdots A^i_{a_i}$ or $A^i = A^i_1 \cdots A^i_{a_i - 1} A^i_*$ be the associated data queries. Let $\mathbb{C}^i = C^i || \mathsf{Tag}^i$ be the ciphertext received for query (N^i, A^i, M^i) . That is, we use superscripts to indicate query numbers and subscripts to denote the block indices in each query.

Let $(N, A, \mathbb{C})$ be the forgery attempt by the adversary, where $N \in \{0,1\}^{|N|}$ is the nonce, $A = A_1 \cdots A_a$ or $A = A_1 \cdots A_{a-1} A_*$ is the associate data, $\mathbb{C} = C || \mathsf{Tag}$ is the ciphertext where $C = C_1 \cdots C_\ell$ (where $|C_i| = m$ for $1 \le i \le \ell$) or $C = C_1 \cdots C_{\ell-1} C_*$ (where $|C_i| = m$ for $1 \le i \le \ell - 1$ and $|C_*| < m$), and $\mathsf{Tag} = (\mathsf{Tag}_e \oplus \mathsf{Tag}_a)[n-1 \cdots n-\tau] \in \{0,1\}^\tau$ is the tag. Let $M = M_1 \cdots M_\ell$ or $M = M_1 \cdots M_{\ell-1} M_*$ denote the corresponding decrypted messages, respectively. Note that no superscripts are used for the strings in the alleged forgery by the adversary. We have the following disjoint cases:

Case 1: $N \notin \{N^1, \cdots N^{q_e}\}$. Adversary has to find a correct Tag that is the first τ bits of the value $\widetilde{R}^{\langle N,x,y \rangle}(\text{final input}) \oplus \mathsf{Tag}_a$ but has not seen any image under $\widetilde{R}^{\langle N,x,y \rangle}(.)$, hence the probability that the adversary can succeed in doing this is $2^{-\tau}$. By "final input" we mean $H_\ell || M_\ell$ or $H_\ell || M_* || 10^{m-|M_*|-1}$ when $|C| \neq 0$ in which case the final tweak used to generate Tag_e will be either $\langle N, \ell, 1 \rangle$ or $\langle N, \ell, 2 \rangle$ (depending on whether the final block is a full block or not); otherwise (i.e. for empty message) the "final input" will be $H_0 || \langle \tau \rangle_m$ and hence the final tweak used to generate Tag_e will be $\langle N, 1, 0 \rangle$.

Case 2: $N = N^i$, $|C| \neq |C^i|$, and one of $|C|$ and $|C^i|$ is a multiple of m but the other is not. We can ignore all queries other than the ith query since the responses to such queries are random and unrelated (because of using different nonces) to the adversary's task to make the alleged forgery $N, A, \mathbb{C}$ with $N = N^i$. That is, we can assume that adversary has only made a single encryption query (N^i, A^i, M^i) and received $C^i || \mathsf{Tag}^i$. Then as in Case 1 the adversary has to find a correct Tag, i.e. the first τ bits of the value $\widetilde{R}^{\langle N,x,y \rangle}(\text{final input}) \oplus \mathsf{Tag}_a$, but has not seen any image under $\widetilde{R}^{\langle N,x,y \rangle}(.)$. Note that we can even give Tag_a to the adversary. More precisely, consider the case that $|C^i|$ is a multiple of m but $|C|$ is not; then adversary must guess the first τbits of the value $\widetilde{R}^{\langle N,\ell,2 \rangle}(\text{final input}) \oplus \mathsf{Tag}_a$, but has seen no image under $\widetilde{R}^{\langle N,\ell,2 \rangle}(.)$. Similarly, in the case that $|C|$ is a multiple of m but $|C^i|$ is not, the adversary must guess the first τ bits of the value $\widetilde{R}^{\langle N,x,y \rangle}(\text{final input}) \oplus \mathsf{Tag}_a$ for $(N, x, y) = (N, 1, 0)$ if $|C| = 0$ or $(N, x, y) = (N, \ell, 1)$ if $|C| \neq 0$, but the adversary has seen no image under $\widetilde{R}^{\langle N,x,y \rangle}(.)$ under either case.Therefore, the probability that the adversary can succeed in guessing Tag is $2^{-\tau}$.

Case 3: $N = N^i$, $|C| \neq |C^i|$, and either both $|C|$ and $|C^i|$ are multiple of m or none of them is. We may ignore all queries but the ith query as responses to such queries are unrelated to the adversary's task at hand. If both $|C|$ and $|C^i|$ are multiple of m then $|C| \neq |C^i|$ means that $\ell \neq \ell^i$, so from (the top of) Figure 3 it can be easily seen that in this case even if the adversary knows Tag_a it must still guess the first τ bits of the output of the random function $\widetilde{R}^{\langle N,\ell,1 \rangle}$ while it has seen no image of this function; the probability to succeed in guessing Tag is clearly $2^{-\tau}$. Now, let's consider the case that neither $|C|$ nor $|C^i|$ is a multiple of m then $|C| \neq |C^i|$ means that we have two cases: (1) $\ell \neq \ell^i$, and (2) $\ell = \ell^i$ but $|C_*| \neq |C^i_*|$. In the first case, it can be seen the adversary must guess the first τ bits of the random function $\widetilde{R}^{\langle N,\ell,2 \rangle}$ while has seen no image of this function; the chance to do so is clearly $2^{-\tau}$. In the second case, the adversary must guess the first τ bits of $\widetilde{R}^{\langle N,\ell,2 \rangle}((M_* \oplus C_*) || (M_* || 10^{m-|M_*|-1}))$ while it has seen (τ bits of) a single image of this function for one different domain point, namely $((M^i_* \oplus C^i_*) || (M^i_* || 10^{m-|M^i_*|-1})$; the probability to succeed in this case is again $2^{-\tau}$. (Note that $|M_*| = |C_*|$. Using 10^* padding for processing messages whose length is not a multiple of m is essential for this part.)

Case 4: $N = N^i$, $|C| = |C^i|$, and $A \neq A^i$. We can ignore all queries except the ith query because the responses to such queries are random and unrelated to the adversary's task to make the alleged forgery

$N, A, \mathbb{C}$ with $N = N^i$. That is, we can assume that adversary has only made a single encryption query (N^i, A^i, M^i) and received $C^i||\mathsf{Tag}^i$. It aims to forge using the same nonce but a different associated data A. The adversary must find a correct $\mathsf{Tag} = (\mathsf{Tag}_e + \mathsf{Tag}_a)[n-1 \cdots n-\tau]$. We consider two subcases: (4a) $|A| \neq 0$ and (4b) $|A| = 0$.

(4a). In this case, let's assume that we even provide the adversary with all the functions $\widetilde{R}^{\langle N,x,y\rangle}(.)$, so that the adversary can compute the correct value of Tag_e. Then the adversary's task will reduce to guessing a correct value for the first τ bits of Tag_a. The only relevant information that the adversary has is the first τ bits of Tag_a^i. We show that even if the whole Tag_a^i is given to the adversary, the chance to correctly guess the first τ bits of Tag_a is still $2^{-\tau}$. This is done by a simple case analysis:

1. if only one of $|A|$ and $|A^i|$ is a multiple of $n+m$ then it is easy to see from Figure 3 that the probability to guess the first τ bits of Tag_a is still $2^{-\tau}$;
2. if $a \neq a_i$ then again from Figure 3 we can see that the probability to guess the first τ bits of Tag_a is $2^{-\tau}$;
3. otherwise, we have $a = a_i$ and either both $|A|$ and $|A^i|$ are multiple of $n+m$ or neither of them is a multiple of $n+m$. These two cases are similar. Let's consider the first one. As we have $A \neq A^i$ then it must be the case that for some j we have $A_j \neq A_j^i$. So, the jth value xored to Tag_a , i.e. $\widetilde{R}^{\langle j,0\rangle}(A_j)$ is a fresh n-bit random value; hence the adversary's chance to guess the first τ bits of Tag_a is $2^{-\tau}$.

(4b). In this case the adversary has seen $C^i||\mathsf{Tag}^i$, where $\mathsf{Tag}^i = (\mathsf{Tag}_e^i \oplus \mathsf{Tag}_a^i)[n-1 \cdots n-\tau]$. To get the forged tuple $(N, \varepsilon, C||\mathsf{Tag})$ be accepted and decrypted, it must find the value of $\mathsf{Tag} = \mathsf{Tag}_e[n-1 \cdots n-\tau]$ (as $\mathsf{Tag}_a = 0^n$ in this case). Now let's give the adversary all functions $\widetilde{R}^{\langle N,x,0\rangle}(.)$ for $1 \leq x \leq \ell$. Even in this case, the adversary has seen no image of the function $\widetilde{R}^{\langle N,x,j\rangle}(.)$ for $j \in \{1,2\}$, since the value $\mathsf{Tag}^i = \mathsf{Tag}_e^i \oplus \mathsf{Tag}_a^i$ that adversary has seen does not reveal any information about Tag_e^i noting that Tag_a^i is random and unrevealed to the adversary. So, the probability that the adversary can correctly guess the first τ bits of $\mathsf{Tag}_e = \widetilde{R}^{\langle N,\ell,j\rangle}(\text{final input})$ for $j = \{1,2\}$ is $2^{-\tau}$. (Note that $j = 1$ when $|C|$ is a multiple of m and $j = 2$ when $|C|$ is not a multiple of m).

Case 5: $N = N^i$, $A = A^i$, and $|C| = |C^i| = \ell m$ is a multiple of m. We can again ignore all queries except the ith query. Let's assume that we make all functions $\widetilde{R}^{\langle x,y\rangle}$ (for $x \geq 1$ and $y \in \{0,1\}$) used in processing the associate data public to the adversary; i.e assume that the adversary even knows the values of Tag_a and Tag_a^i. Now remember that the adversary must not repeat the known tuple $(N^i, A^i, C^i||\mathsf{Tag}^i)$ as its decryption query, so it must be the case that $C \neq C^i$ as otherwise any $\mathsf{Tag} \neq \mathsf{Tag}^i$ will be incorrect and rejected. Therefore, we may assume that the alleged forgery will be of the form $(N, A, C||\mathsf{Tag})$ such that $C_j \neq C_j^i$ for some $1 \leq j \leq \ell$. Now referring to (the top of) Figure 3 it is easy to see that if $C_\ell \neq C_\ell^i$ then the probability that the adversary can correctly guess the value of Tag is $2^{-\tau}$; otherwise there are two cases: (1) if $H_\ell \neq H_\ell^i$ the chance that Tag is correct is $2^{-\tau}$; (2) if the event $H_\ell = H_\ell^i$ happens then adversary can simply use $\mathsf{Tag} = \mathsf{Tag}^i$, but this event only happens with probability at most $\ell 2^{-n}$ noting that $|H_i| = n$ (note that we credit the adversary for any possible collision in the iteration, there are ℓ blocks and the probability of each collision under the random function is 2^{-n}). So, the total success probability in this case is bounded by $\frac{1}{2^\tau} + \frac{\ell}{2^n}$.

Case 6: $N = N^i$, $A = A^i$, and $|C| = |C^i|$ is not a multiple of m. It is easy to see from Figure 3 that the analysis of this case is the same as that of Case 5 and the success probability of the adversary is bounded by $\frac{1}{2^\tau} + \frac{\ell}{2^n}$.

Finally, using the results of Bellare et al. [5] we get the bound against adversaries that make q_v decryption (verification) queries as $\frac{q_v}{2^\tau} + \frac{q_v \ell}{2^n}$.

5.2 Instantiating Tweakable RFs with PRFs

We proceed to complete the proof of Theorem 1 in two steps.

5.2.1 Step 1.

Replace the tweakable RF $\widetilde{R} : \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ in $\mathbb{OMD}$ with a tweakable PRF $\widetilde{F} : \mathcal{K} \times \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$, where $\mathcal{K} = \{0,1\}^k$. The following lemma states the classical bound on the security loss induced by this replacement step. The proof is a straightforward reduction and omitted here.

Lemma 3 *Let $\widetilde{R} : \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be a tweakable RF and $\widetilde{F} : \mathcal{K} \times \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be a tweakable PRF. Then*

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{priv}}_{\mathbb{OMD}[\widetilde{F},\tau]}(t, q_e, \sigma_e, \ell_{max}) &\leq \mathbf{Adv}^{\mathrm{priv}}_{\mathbb{OMD}[\widetilde{R},\tau]}(q_e, \sigma_e, \ell_{max}) + \mathbf{Adv}^{\widetilde{\mathrm{prf}}}_{\widetilde{F}}(t', \sigma_e) \\
\mathbf{Adv}^{\mathrm{auth}}_{\mathbb{OMD}[\widetilde{F},\tau]}(t, q_e, q_v, \sigma, \ell_{max}) &\leq \mathbf{Adv}^{\mathrm{auth}}_{\mathbb{OMD}[\widetilde{R},\tau]}(q_e, q_v, \sigma, \ell_{max}) + \mathbf{Adv}^{\widetilde{\mathrm{prf}}}_{\widetilde{F}}(t'', \sigma)
\end{aligned}$$

where q_e and q_v are, respectively, the number of encryption and decryption queries, $q = q_e + q_v$, ℓ_{max} denotes the maximum number of m-bit blocks in an encryption or decryption query, $t' = t + cn\sigma_e$ and $t'' = t + c'n\sigma$ for some constants c, c', and σ_e and σ are the total number of calls to the underlying compression function F in all queries asked by the CPA and CCA adversaries against the privacy and authenticity of the scheme, respectively.

5.2.2 Step 2.

We instantiate a tweakable PRF using a PRF by means of XORing (part of) the input by a mask generated as a function of the key and tweak as shown in Fig. 4. This method to tweak a PRF is (essentially) the XE method of [19]. In OMD the tweaks are of the form $T = (\alpha, i, j)$ where $\alpha \in \mathcal{N} \cup \{\varepsilon\}$, $1 \leq i \leq 2^{n-8}$ and $j \in \{0,1,2\}$. We note that not all combinations are used; for example, if $\alpha = \varepsilon$ (empty) which corresponds to processing of the associate data in Figure 1 then $j \neq 2$. The masking function $\Delta_K(T) = \Delta_K(\alpha, i, j)$ outputs an n-bit mask such that the following two properties hold for any fixed string $H \in \{0,1\}^n$:

1. $\Pr[\Delta_K(\alpha, i, j) = H] \leq 2^{-n}$ for any (α, i, j)
2. $\Pr[\Delta_K(\alpha, i, j) \oplus \Delta_K(\alpha', i', j') = H] \leq 2^{-n}$ for $(\alpha, i, j) \neq (\alpha', i', j')$

where the probabilities are taken over random selection of the secret key K.

It is easy to verify that these two properties are satisfied by the specific masking scheme of OMD as described in Section 4.

Lemma 4 *Let $F : \mathcal{K} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be a function family with key space $\mathcal{K}$. Let $\widetilde{F} : \mathcal{K} \times \mathcal{T} \times (\{0,1\}^n \times \{0,1\}^m) \to \{0,1\}^n$ be defined by $\widetilde{F}_K^{\langle T \rangle}(X||Y) = F_K((X \oplus \Delta(T))||Y)$ for every $T \in \mathcal{T}, K \in \mathcal{K}$, $X \in \{0,1\}^n, Y \in \{0,1\}^m$ and $\Delta_K(T)$ is the masking function of OMD as defined in Section 4. If F is PRF then $\widetilde{F}$ is tweakable PRF; more precisely*

$$\mathbf{Adv}^{\widetilde{\mathrm{prf}}}_{\widetilde{F}}(t, q) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(t', 2q) + \frac{3q^2}{2^n}$$

.

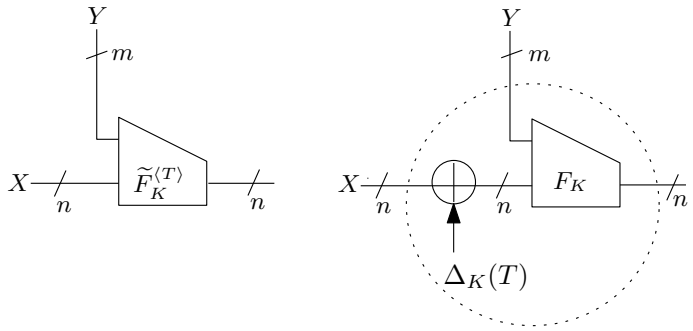

Figure 4: Building a tweakable PRF $\widetilde{F}_K^{\langle T \rangle} : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ using a PRF $F_K : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$. There are several efficient ways to define the masking function $\Delta(T)$ [12,17,19]; we use the method of [17].

The proof is a simple adaptation of a similar result on the security of the XE construction (to tweak a blockcipher) in [17]. As we use a PRF rather than PRP, our bound has two main terms. The first term is a single birthday bound loss of $\frac{0.5q^2}{2^n}$ to take care of the case that a collision might happen when computing the initial mask $\Delta_{N,0,0} = F_K(N||10^{n-1-|N|}, 0^m)$ using a PRF (F) rather than a PRP (as in [17]). The analysis of the remaining term (i.e. $\frac{2.5q^2}{2^n}$) is essentially the same as the similar part in [17], but we note that in the context of our construction as we are directly dealing with PRFs unlike [17] in which PRPs are used, the bound obtained here does not have any loss terms caused by the switching (PRF<>PRP) lemma. Therefore, instead of the $\frac{6q^2}{2^n}$ bound in [17] (from which $\frac{3.5q^2}{2^n}$ is due to using the switching lemma) our bound has only $\frac{2.5q^2}{2^n}$.

6 Features

Using only a single well-known primitive. OMD is designed as a mode of operation for a keyed compression function. Together with blockciphers and permutations, compression functions are among the most well-known and widely used symmetric key primitives. We have a rich source of secure compression functions thanks to more than two decades of public research and standardization activities on hash functions.

Provable security based on a single widely-accepted standard assumption. The security goals of privacy and authenticity for OMD are achieved provably in the sense of reduction-based cryptography; that is, any attack against these security goals will imply an attack against the classical PRF property of the underlying compression function. We note that any keyed compression function (either a dedicated-key one or keyed via some part of its input) must provide the classical PRF property when its key is secret as otherwise it will be considered useless for any secret key application, e.g. for being used as a MAC. That is, the base PRF assumption on the compression function upon which the security of OMD relies is highly assured for compression functions of the practical, standard hash functions, thanks to the vast amount of cryptanalytic work on these functions.

Requiring minimal basic operations in addition to the core primitive. The only operations that OMD needs *in addition to* its core compression function are the basic operations of bitwise xoring two binary strings and shifting a binary string.

Integrated (one-pass) AEAD scheme. In OMD the mechanisms for providing privacy and authenticity of the message are coupled in a single pass of (a variant of) the Merkle-Damgård iteration of the compression function. This is aimed to make OMD as much efficient as possible (up to the limits that are inherent to any compression function based AEAD scheme).

Online Encryption. OMD encryption is online; that is, it outputs a stream of ciphertext as a stream of plaintext arrives with a constant latency and using constant memory. After receiving an indication that the plaintext is over, the final part of ciphertext together with the tag is output.

Internally Online Decryption. OMD decryption is *internally* online: one can generate a stream of plaintext bits as the stream of ciphertext bits comes in, but no part of the plaintext stream will be revealed before the whole ciphertext stream is decrypted and the tag is verified to be correct. That is, nothing about the decrypted plaintext should be made available to adversaries if the tag is incorrect signifying that the queried ciphertext is invalid.

Flexible key size. OMD-sha256 can support any key length between 80 bits and 256 bits. This will be useful for applications requiring unconventional key lengths, e.g. 96-bit keys.

Efficient. If implemented with a member of the SHA family, OMD can take advantage of the newly introduced Intel® instructions that support performance acceleration of the Secure Hash Algorithm (SHA) on Intel® Architecture processors. In particular, our main recommended scheme for CAESAR, called OMD-sha256, is aimed to get the most out of these new performance accelerating instructions.

Resistance against software-level timing attacks. Most AES software implementations risk leaking their keys through cache timing [10] unless they are implemented on machines with Intel® CPUs supporting the constant-time AES-NI and PCLMULQDQ instructions. In comparison, we note that the only operations in OMD-sha256 are: bitwise XOR, AND and OR of two binary strings (32-bit words in the compression function of SHA-256 and 256-bit words in the OMD iteration), fixed-distance (left and right) shift of a binary string (32-bit words in the compression function of SHA-256 and 256-bit words in the OMD iteration), and 32-bit addition (of words in the compression function of SHA-256). These operations have the virtue of taking constant time on typical CPUs in which case the implementations can avoid software-level timing based side-channel leaks.

7 Design Rationale

The main design rationales behind OMD are the following:

Provable security. We aimed to have a scheme with a sound security guarantee in the style of reduction-based provable security relying only on a single well-established standard assumption on the underlying primitive, namely the PRF assumption on the keyed compression function. The security goals of privacy and authenticity for OMD are achieved provably; that is, any attack against these security goals will imply an attack against the classical PRF property of the underlying compression function. We note that any good keyed compression function (either a dedicated-key one or keyed via some part of its input) must provide the classical PRF property when its key is secret as otherwise it will be considered useless for almost any secret key application, e.g. for being used as the compression function of a hash function in the standard HMAC algorithm. That is, the base PRF assumption on the compression function upon which the security

of OMD relies is highly assured for compression functions of the practical, standard hash functions, thanks to the vast amount of cryptanalytic work on these functions.

Simple structure. Simplicity is important in any cryptographic algorithm: the easier an algorithm is to understand, the easier it is to analyze and to get confidence on its security, and also less prone it is to implementation errors. Therefore, simplicity was one of our core design goals. The high level structure of OMD is quite simple and resembles the well-known structures for hash functions and MACs, namely, the part that is processing the message resembles the Merkle-Damgård iteration where at each iteration random bits are derived from the chaining values to be used for encryption and a key-dependent offset value is xored to the chaining values. The part for processing the associated data is inspired by the XMACC scheme(counter-based XOR MAC scheme) [6] and is a simple adaptation of the similar hashing process in the OCB3 algorithm [17]. We note that when the message is empty then OMD acts almost the same as XMACC on the associated data.

No trapdoor. The designers have not hidden any weaknesses in this cipher. Any attack against security of OMD means an attack against the specific compression function that is used for instantiating OMD. For example, attacking OMD-sha256 will imply attacking the compression function of SHA-256 in the PRF sense.

8 Intellectual Property

The owners of OMD are the same as the designers and submitters of the algorithm as listed on the first page of this submission. There are no patents, patent applications, planned patent applications, or other intellectual-property constraints relevant to use of OMD. If any of this information changes, the submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

9 Consent

The submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

References

[1] Secure Hash Standard (SHS) . NIST FIPS PUB 180-4 (Mar 2012)

[2] Intel® SHA Extensions (Jul 2013), http://software.intel.com/en-us/articles/intel-sha-extensions

[3] Bellare, M.: New Proofs for NMAC and HMAC: Security Without Collision-Resistance. IACR Cryptology ePrint Archive 2006, 43 (2006)

[4] Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A Concrete Security Treatment of Symmetric Encryption. In: FOCS. pp. 394–403 (1997)

[5] Bellare, M., Goldreich, O., Mityagin, A.: The Power of Verification Queries in Message Authentication and Authenticated Encryption. IACR Cryptology ePrint Archive 2004, 309 (2004)

[6] Bellare, M., Guérin, R., Rogaway, P.: XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions. In: Coppersmith, D. (ed.) CRYPTO. LNCS, vol. 963, pp. 15–28. Springer (1995)

[7] Bellare, M., Namprempre, C.: Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 531–545. Springer (2000)

[8] Bellare, M., Namprempre, C.: Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. J. Cryptology 21(4), 469–491 (2008)

[9] Bellare, M., Rogaway, P.: Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 317–330. Springer (2000)

[10] Bernstein, D.J.: Cache-timing attacks on AES (2005), http://cr.yp.to/papers.html#cachetiming

[11] Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password Interception in a SSL/TLS Channel. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 583–599. Springer (2003)

[12] Chakraborty, D., Sarkar, P.: A General Construction of Tweakable Block Ciphers and Different Modes of Operations. IEEE Transactions on Information Theory 54(5), 1991–2006 (2008)

[13] Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Remark on variable tag lengths and OMD. CAESAR competition mailing list, 25 April 2014

[14] Fleischmann, E., Forler, C., Lucks, S.: McOE: A Family of Almost Foolproof On-Line Authenticated Encryption Schemes. In: Canteaut, A. (ed.) FSE. LNCS, vol. 7549, pp. 196–215. Springer (2012)

[15] Iwata, T., Ohashi, K., Minematsu, K.: Breaking and Repairing GCM Security Proofs. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO. LNCS, vol. 7417, pp. 31–49. Springer (2012)

[16] Katz, J., Yung, M.: Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 284–299. Springer (2001)

[17] Krovetz, T., Rogaway, P.: The Software Performance of Authenticated-Encryption Modes. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 306–327. Springer (2011)

[18] Rogaway, P.: Authenticated-Encryption with Associated-Data. In: ACM Conference on Computer and Communications Security. pp. 98–107 (2002)

[19] Rogaway, P.: Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: ASIACRYPT. pp. 16–31 (2004)

[20] Rogaway, P.: Nonce-Based Symmetric Encryption. In: Roy, B.K., Meier, W. (eds.) FSE. LNCS, vol. 3017, pp. 348–359. Springer (2004)

[21] Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In: ACM Conference on Computer and Communications Security. pp. 196–205 (2001)

[22] Rogaway, P., Shrimpton, T.: A Provable-Security Treatment of the Key-Wrap Problem. In: EUROCRYPT. pp. 373–390 (2006)

[23] Vaudenay, S.: Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–546. Springer (2002)

A Changes from version 1.0

The specification of OMD for the 2nd round of the CAESAR competition received a minor tweak in the initialization stage of computing the masking values; namely, the value L_* used to compute the $\Delta_{N,i,j}$ and $\bar{\Delta}_{i,j}$ masking offsets is now defined to be $L_* = F_K(0^n, \langle\tau\rangle_m)$ (instead of $L_* = F_K(0^n, 0^m)$ in version 1.0) where $\langle\tau\rangle_m$ is the length of the authentication tag in bits, represented as an m-bit string. The changes in the submission documentation from version 1.0 that correspond to this tweak are on page 12 (in the **Initialization** step for computing the masking values) and page 14 (in **Algorithm** INITIALIZE in Fig. 2).

We remark this change in the way L_* is computed has no impact on the existing security analysis. As the tag length τ is defined to be a *parameter* of OMD, it will be a constant for every instance. Moreover, because the nonce is always padded to a non-zero block, the inputs to the compression function F_K used to derive $\Delta_{N,0,0}$ and $(0^n, \langle\tau\rangle_m)$ can never collide. These two properties are sufficient for the analysis to carry over as long as $\log_2(n) < m$, which is a requirement that is met in any practical setting.

The reason for introducing this tweak was a possible *misuse scenario* mentioned by Dobraunig et al. [13], in which an adversary can interact with several instances of OMD that share all the parameters except the tag length, while using *the same secret key.* Although it was labeled as an "attack" by Dobraunig et al., we noted that this phenomenon is not an attack per se because it assumes that OMD is used in a way that does not comply with the specification (i.e. incorrectly). Nevertheless, considering this as a type of misuse scenario, the simple tweak we propose for OMD version 2.0 prevents any tag-length misusing attack of this kind by making every call to the compression function depend on τ.