The CWC Authenticated Encryption (Associated Data) Mode

Tadayoshi Kohno
UC San Diego
9500 Gilman Drive, MC 0114
La Jolla, CA 92093
tkohno@cs.ucsd.edu

John Viega
Virginia Tech
6066 Leesburg Pike, Suite 500
Falls Church, VA 22041
viega@securesoftware.com

Doug Whiting
Hifn, Inc.
5973 Avenida Encinas, Suite 110
Carlsbad, CA 92009
dwhiting@hifn.com

May 27, 2003

Abstract

We introduce CWC, a new block cipher mode of operation designed to protect both the privacy and the authenticity of encapsulated data. Important properties of CWC include:

1. **Performance.** CWC is **parallelizable** and is **efficient** in both hardware and software.
2. **Security.** CWC is **provably secure** and its provable security depends only on the pseudorandomness of the underlying block cipher. No other cryptographic primitives are used and no other assumptions are made.
3. **Patent-free.** To the best of our knowledge CWC is **not covered by any patents**.

CWC is currently the only dedicated authenticated encryption with associated data (AEAD) scheme that simultaneously has these three properties (e.g., CCM and EAX are not parallelizable and OCB is not patent-free). Having all three of these properties makes CWC a strong candidate for use with future high-performance systems.

Contents

1 Introduction

There has recently been significant interest in developing block cipher modes of operation capable of simultaneously protecting both the *privacy* and the *authenticity/integrity* of encapsulated data. Such modes of operation are often called *authenticated encryption (AE) schemes* or, if the schemes are capable of authenticating more data than they encrypt, *authenticated encryption with associated data (AEAD) schemes*.

In this work we propose a dedicated AEAD scheme that is **patent-free**, **provably-secure**, **parallelizable** and **efficient** in both hardware and software. To our knowledge, ours is the only dedicated AEAD scheme that simultaneously has all of these properties. (Our construction also has other desirable properties. For example, it is clean and simple (in our opinion), on-line, uses a single block cipher key, and allows for pre-processing of associated data or other header fields.)

Our general construction, **CWC**, is based on what is called "the Encrypt-then-Authenticate generic composition paradigm." In particular, CWC essentially combines a Carter-Wegman message authentication scheme [18] with CTR mode encryption in an Encrypt-then-Authenticate manner. The general idea is as follows: given a pair of strings (A, M) and a nonce N as input, the CWC encapsulation algorithm encrypts M with CTR mode to get some intermediate ciphertext σ. It then uses a Carter-Wegman MAC and the nonce N to MAC the pair (A, σ). If we let τ denote the resulting MAC tag, then the output of the CWC encapsulation algorithm is the concatenation of σ and τ. CWC is designed to protect the privacy of M and the integrity of both A and M. Although based on the Encrypt-then-Authenticate generic composition paradigm, CWC is not a generic composition construction; for example, the CWC encryption and MAC components share the same block cipher key. This means, among other things, that we had to prove the security of CWC directly, rather than invoke previous results about the generic composition paradigm.

Let us begin by discussing some of the motivations for CWC.

WHY DO WE WANT DEDICATED AUTHENTICATED ENCRYPTION SCHEMES? The traditional approach to achieving authenticated encryption is to combine some standard encryption scheme (e.g., CBC mode) with some standard message authentication scheme (e.g., HMAC). This is known as the *generic-composition* approach and was first explored in [1] and [9]. Unfortunately, such generic-composition constructions are often *ad hoc* and, as illustrated in [1] and [9], it is very easy to accidentally combine secure encryption schemes with secure MACs and still get insecure authenticated encryption schemes.

One of the biggest advantages of dedicated AEAD schemes over generic-composition AEAD schemes is that dedicated AEAD schemes are not prone to such accidental errors. That is, since dedicated AEAD schemes clearly specify how to achieve both privacy and authenticity, there is no longer the risk of someone accidentally combing a privacy/encryption component with an authenticity/MAC component in an insecure fashion. Furthermore, since most applications that require privacy also require integrity, it is logical to focus on tools capable of providing both services simultaneously. There is thus great value in developing and standardizing dedicated AEAD schemes, as evidenced by a wealth of papers in this area [2, 4, 6, 7, 14, 15, 19].

PATENTS. Pragmatically, patents are a major impediment to the standardization and wide-spread deployment of some of the modes presented in the above-mentioned papers. In particular, three independent parties have applied for patents on single-pass authenticated encryption schemes. It is not our purpose to describe the specifics of these patent applications (and, indeed, the specifics are not completely known to the public). Rather, we point out that the existence of these patent applications makes many existing authenticated encryption modes less attractive, and therefore less amenable to standardization and deployment. To exemplify this point, we note that although

Rogaway, Bellare, Black, and Krovetz's OCB mode [15] is very efficient and elegant, it was apparently rejected from the IEEE 802.11 wireless working group largely because of the fact that it was covered by patent applications from multiple parties.

WHAT IS NEEDED? Noting the need for patent-free dedicated AEAD schemes, Whiting, Ferguson, and Housley proposed a patent-free AEAD scheme called CCM [19] which, apparently because of its patent-free nature, has been adopted by the IEEE 802.11 working group. CCM was recently followed by another construction, called EAX, by Bellare, Rogaway, and Wagner [2]. Since CCM and EAX are based on the generic-composition approach (they both essentially combine standard CTR mode encryption with variants of CBC-MAC message authentication), CCM and EAX do not fall under the aforementioned patent applications.

There is, however, one significant disadvantage with both CCM and EAX: the CCM and EAX encryption and decryption operations are not parallelizable. That is, although the CTR mode portions of CCM and EAX are clearly parallelizable, their CBC-MAC portions are not. Parallelizability is, however, very important. For example, without the ability to parallelize the encryption process, using current technology it does not seem possible to build a single hardware engine for CCM or EAX capable of encrypting beyond approximately 2 Gbps.[1] Although 2 Gbps might be adequate for today's applications, such speeds will not be adequate for the coming 10 Gbps network devices.

Therefore, there is a need for a patent-free dedicated mode of operation capable of encrypting and authenticating data at 10 Gbps.

THE CWC SOLUTION. We propose a general paradigm, called CWC, that addresses all the aforementioned issues. In particular, our preferred instantiation of CWC for 128-bit block ciphers is un-patented, provably-secure, parallelizable, and efficient in both hardware and software. The parallelizability enables high-speed CWC hardware implementations to encrypt at 10 Gbps when using AES.

Throughout the body of this paper we will focus on our instantiation of the CWC paradigm for 128-bit block ciphers.[2] In particular, we focus on CWC-AES-kl, a CWC instantiation with AES-kl as the underlying block cipher (here AES-kl denotes kl-bit AES, where the key length $\mathsf{kl} \in \{128, 192, 256\}$). When our results apply to AES with all key lengths, we shall simply refer to CWC-AES. Instead of writing CWC-AES-kl for some appropriate kl, we shall write CWC-BC or simply CWC when we mean the general CWC paradigm instantiated like CWC-AES-kl but with any 128-bit block cipher BC in place of AES-kl.

Note the difference in font between CWC, the general paradigm, and CWC, our specific proposal.

ACHIEVING PARALLELISM. Clearly the CTR mode portion of CWC is parallelizable. Furthermore, the core of the Carter-Wegman MAC portion of CWC (a.k.a. the universal hashing portion of CWC) can be made parallelizable. In the case of CWC, the universal hashing step works by computing

$$Y_1 x^n + Y_2 x^{n-1} + Y_3 x^{n-2} + Y_4 x^{n-3} + \cdots + Y_n x + Y_{n+1} \bmod 2^{127} - 1 \ .$$

where $Y_1, \ldots, Y_{n+1}$ are 96-bit integers[3] corresponding to the pair (A, σ) and x is an integer modulo the prime $2^{127} - 1$. It is well-known that the computation of this polynomial is parallelizable. For

[1] It is always possible to build two totally independent units and process two packets at a time, but this is dramatically more complex, requiring twice the area, plus a load balancer.

[2] If desired, it is possible to instantiate the general CWC paradigm with 64-bit block ciphers, although certain limitations (e.g., nonce size) apply to such variants. We do not present a 64-bit CWC variant here since we are primarily concerned with new, high-speed systems using AES, not legacy applications.

[3] Actually, Y_{n+1} may be more than 96-bits long, but we ignore that detail here.

example, if we have two processors available, we can rewrite the above polynomial as

$$\Big(Y_1 y^m + Y_3 y^{m-1} + \cdots + Y_n\Big)x + \Big(Y_2 y^m + Y_4 y^{m-1} + \cdots + Y_{n+1}\Big) \bmod 2^{127} - 1 \ ,$$

where $y = x^2 \bmod 2^{127} - 1$, $m = (n-1)/2$, and we assume for illustrative purposes that n is odd. We can then compute both the left and the right portions of the above in parallel. Additional parallelism can be achieved by further splitting the original polynomial into j polynomials in $y' = x^j \bmod 2^{127} - 1$.

Single key. The CWC paradigm uses a single block cipher key K. The key K is used in all applications of the underlying block cipher and is used to derive a subkey K_h for use in CWC's universal hashing step. In the case of our CWC instantiation, deriving K_h requires one block cipher invocation. The main advantage with deriving subkey K_h from K is that it simplifies key management and reduces the costs associated with fetching key material in hardware, which can be a bottleneck.

Performance. Let (A, M) be some input to the CWC encapsulation algorithm (recall that A is the associated data and M is the message to encrypt). Assuming that the universal hashing subkey is maintained across invocations, encapsulating (A, M) takes $\lceil |M|/128 \rceil + 2$ block cipher invocations. The polynomial used in CWC's universal hashing step will have degree $d = \lceil |A|/96 \rceil + \lceil |M|/96 \rceil$. There are several ways to evaluate this polynomial (details in Section 5). For example, assuming no precomputation, we could evaluate this polynomial using d 128-by-128-bit multiplies. As another example, assuming n precomputed powers of the hash subkey, which are cheap to maintain in software for reasonable n, we could evaluate the polynomial using $d - m$ 96-by-128-bit multiplies and m 128-by-128-bit multiplies, where $m = \lceil (d+1)/n \rceil - 1$.

As noted before, it is possible to implement CWC-AES in hardware at 10 Gbps using conventional ASIC technology. Specifically, at 0.13 micron, it should take about 300 Kgates to reach 10 Gbps throughput. In software, experimental results show that the current best implementation of CWC-AES-128 on a Pentium III (due to Brian Gladman) runs significantly faster than CCM and EAX with 128-bit AES and implemented with popular crypto libraries. Using the precomputation approach from Bernstein [3], we anticipate reducing the cost of CWC's universal hashing step to around 8 cpb, thereby significantly improving the performance of CWC-AES in software.

Provable security. CWC is a provably-secure AEAD scheme assuming that the underlying block cipher is a secure pseudorandom function or pseudorandom permutation. Consequently, if we believe AES to be a secure pseudorandom permutation (which is a widely-held belief), then CWC-AES is secure. For our proofs of security, we use Rogaway's AEAD notions from [14]. In our provable security results we clearly show that the same block cipher key can be used in CWC's CTR mode portion, in the generation of the hash subkey K_h, and in the block cipher applications used within CWC's message authentication portion.

1.1 Background and related work

The notion of an *authenticated encryption (AE) scheme* was formalized in Katz–Yung [7] and Bellare–Namprempre [1] and the notion of an *authenticated encryption with associated data (AEAD) scheme* was formalized in Rogaway [14]. In [1, 9], Bellare–Namprempre and Krawczyk explored ways to combine standard encryption schemes with MACs to achieve authenticated encryption. A number of dedicated AE and AEAD schemes also exist, including RPC [7], XCBC [4], IACBC [6], OCB [15], CCM [19], and EAX [2]. Within the scope of dedicated block cipher-based AEAD schemes, CWC's closest relatives are CCM and EAX, which also use two passes and are unpatented. From a broader perspective, CWC is similar to the combination of McGrew's UST [12] and TMMH [11], where one of the main advantages of CWC over UST+TMMH is CWC's small key size, which can be a bottleneck for UST+TMMH in hardware at high speeds.

Rogaway and Wagner recently released a critique of CCM [16]. For each issue raised in [16], we find that we have already addressed the issue (e.g., we designed CWC to be on-line) or we disagree with the issue (e.g., we feel that it is sufficient for new modes of operation to handle arbitrary octet-length, as opposed to arbitrary bit-length, messages[4]).

The integrity portion of CWC builds on top of the Carter-Wegman universal hashing approach to message authentication [18]. Like Bernstein's hash127 [3], CWC's universal hash function evaluates a polynomial over the integers modulo the prime $2^{127}-1$. The main difference between hash127 and the CWC universal hash function is that hash127 uses signed 32-bit coefficients and CWC uses unsigned 96-bit coefficients.

In April 2003 we introduced an Internet-Draft, within the IRTF Crypto Forum Research Group, specifying the CWC-AES mode of operation. The latest version of the Internet-Draft can be found at `http://www.zork.org/cwc` or on the IETF website `http://www.ietf.org`.

1.2 Outline

We begin in Section 2 with some preliminaries and then describe the CWC mode of operation in Section 3. In Section 4 we present our formal statements of security for CWC and in Section 5 we discuss the performance of CWC. We close in Section 6.

Appendix A presents a summary of CWC's properties. Appendix B contains our intellectual property statement. Appendix C contains the formal proofs of security for CWC, as well as a description of our general CWC paradigm. Appendix D contains test vectors.

2 Preliminaries

Notation. If x is a string then $|x|$ denotes its length in *bits* (not octets). Let ε denote the empty string. If x and y are two equal-length strings, then $x \oplus y$ denotes the xor of x and y. If X and Y are sets, then $\mathsf{Func}(X, Y)$ denotes the set of all functions from X to Y and $\mathsf{Perm}(X)$ denotes the set of all permutations on X. If l and L are positive integers, then $\mathsf{Func}(l, L)$ denotes the set of all functions from $\{0,1\}^l$ to $\{0,1\}^L$ and $\mathsf{Perm}(L)$ denotes the set of all permutations on $\{0,1\}^L$.

If N is a non-negative integer and l is an integer such that $0 \le N < 2^l$, then $\mathsf{tostr}(N, l)$ denotes the encoding of N as an l-bit string in big-endian format. If x is a string, then $\mathsf{toint}(x)$ denotes the integer corresponding to string x in big-endian format (the most significant bit is *not* interpreted as a sign bit). For example, $\mathsf{toint}(10000010) = 2^7 + 2 = 130$.

Let $x \leftarrow y$ denote the assignment of y to x. If X is a set, let $x \stackrel{\$}{\leftarrow} X$ denote the process of uniformly selecting at random an element from X and assigning it to x. If f is a randomized algorithm, let $x \stackrel{\$}{\leftarrow} f(y)$ denote the process of running f with input y and a uniformly selected random tape.

When we refer to the time of an algorithm or experiment, we include the size of the code (in some fixed encoding). There is also an implicit big-$\mathcal{O}$ surrounding all time references.

Authenticated encryption schemes with associated data. We use Rogaway's notion of an *authenticated encryption with associated data (AEAD) scheme* [14].

An AEAD scheme $\mathcal{SE} = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ consists of three algorithms and is defined over some key space $\mathsf{KeySp}_{\mathcal{SE}}$, some nonce space $\mathsf{NonceSp}_{\mathcal{SE}} = \{0,1\}^n$, n a positive integer, some associated data

[4] Although we stress that, if desired, it is easy to modify CWC to handle arbitrary bit-length messages. See Remark 3.8.

(header) space $\mathsf{AdSp}_{\mathcal{SE}} \subseteq \{0,1\}^*$, and some payload message space $\mathsf{MsgSp}_{\mathcal{SE}} \subseteq \{0,1\}^*$. We require that membership in $\mathsf{MsgSp}_{\mathcal{SE}}$ and $\mathsf{AdSp}_{\mathcal{SE}}$ can be efficiently tested and that if M, M' are two strings such that $M \in \mathsf{MsgSp}_{\mathcal{SE}}$ and $|M'| = |M|$, then $M' \in \mathsf{MsgSp}_{\mathcal{SE}}$.

The randomized key generation algorithm $\mathcal{K}_e$ returns a key $K \in \mathsf{KeySp}_{\mathcal{SE}}$; we denote this process as $K \xleftarrow{\$} \mathcal{K}_e$. The deterministic encryption algorithm $\mathcal{E}$ takes as input a key $K \in \mathsf{KeySp}_{\mathcal{SE}}$, a nonce $N \in \mathsf{NonceSp}_{\mathcal{SE}}$, a header (or associated data) $A \in \mathsf{AdSp}_{\mathcal{SE}}$, and a payload message $M \in \mathsf{MsgSp}_{\mathcal{SE}}$, and returns a ciphertext $C \in \{0,1\}^*$; we denote this process as $C \leftarrow \mathcal{E}_K^{N,A}(M)$ or $C \leftarrow \mathcal{E}_K(N, A, M)$. The deterministic decryption algorithm $\mathcal{D}$ takes as input a key $K \in \mathsf{KeySp}_{\mathcal{SE}}$, a nonce $N \in \mathsf{NonceSp}_{\mathcal{SE}}$, a header $A \in \mathsf{AdSp}_{\mathcal{SE}}$, and a string $C \in \{0,1\}^*$ and outputs a message $M \in \mathsf{MsgSp}_{\mathcal{SE}}$ or the special symbol $\mathsf{INVALID}$ on error; we denote this process as $M \leftarrow \mathcal{D}_K^{N,A}(C)$. We require that $\mathcal{D}_K^{N,A}(\mathcal{E}_K^{N,A}(M)) = M$ for all $K \in \mathsf{KeySp}_{\mathcal{SE}}$, $N \in \mathsf{NonceSp}_{\mathcal{SE}}$, $A \in \mathsf{AdSp}_{\mathcal{SE}}$, and $M \in \mathsf{MsgSp}_{\mathcal{SE}}$. Let $l(\cdot)$ denote the *length function* of $\mathcal{SE}$; i.e., for all keys K, nonces N, headers A, and messages M, $|\mathcal{E}_K^{N,A}(M)| = l(|M|)$.

Privacy. Let $\$(\cdot,\cdot,\cdot)$ be an oracle that, on input $(N, A, M) \in \mathsf{NonceSp}_{\mathcal{SE}} \times \mathsf{AdSp}_{\mathcal{SE}} \times \mathsf{MsgSp}_{\mathcal{SE}}$, returns a random string of length $l(|M|)$. Let B be an adversary with access to an oracle and that returns a bit. Then

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathcal{SE}}(B) = \Pr\left[\, K \xleftarrow{\$} \mathcal{K}_e \;:\; B^{\mathcal{E}_K(\cdot,\cdot,\cdot)} = 1 \,\right] - \Pr\left[\, B^{\$(\cdot,\cdot,\cdot)} = 1 \,\right]$$

is the IND\$-CPA-*advantage* of B in breaking the privacy of $\mathcal{SE}$ under chosen-plaintext attacks; i.e., $\mathbf{Adv}^{\mathrm{priv}}_{\mathcal{SE}}(B)$ is the advantage of B in distinguishing between ciphertexts from $\mathcal{E}_K(\cdot,\cdot,\cdot)$ and random strings. An adversary B is *nonce-respecting* if it never queries its oracle with the same nonce twice. Intuitively, a scheme $\mathcal{SE}$ preserves privacy under chosen plaintext attacks if the IND\$-CPA-advantage of all nonce-respecting adversaries using reasonable resources is small.

Integrity/authenticity. Let F be a forging adversary and consider an experiment in which we first pick a random key $K \xleftarrow{\$} \mathcal{K}_e$ and then run F with oracle access to $\mathcal{E}_K(\cdot,\cdot,\cdot)$. We say that F *forges* if F returns a pair (N, A, C) such that $\mathcal{D}_K^{N,A}(C) \neq \mathsf{INVALID}$ but F did not make a query (N, A, M) to $\mathcal{E}_K(\cdot,\cdot,\cdot)$ that resulted in a response C. Then

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}}(F) = \Pr\left[\, K \xleftarrow{\$} \mathcal{K}_e \;:\; F^{\mathcal{E}_K(\cdot,\cdot,\cdot)} \text{ forges } \,\right]$$

is the AUTH-*advantage* of F in breaking the integrity/authenticity of $\mathcal{SE}$. Intuitively, $\mathcal{SE}$ preserves integrity if the AUTH-advantage of all nonce-respecting adversaries using reasonable resources is small.

Pseudorandom functions and permutations. Let F be a family of functions from D to R. Let A be an adversary with access to an oracle and that returns a bit. Then

$$\mathbf{Adv}^{\mathrm{prf}}_{F}(A) = \Pr\left[\, f \xleftarrow{\$} F \;:\; A^{f(\cdot)} = 1 \,\right] - \Pr\left[\, g \xleftarrow{\$} \mathsf{Func}(D, R) \;:\; A^{g(\cdot)} = 1 \,\right]$$

denotes the PRF-advantage of A in distinguishing a random instance of F from a random function. Intuitively, we say that F is a secure PRF if the PRF-advantages of all adversaries using reasonable resources is small.

Let F be a a family of functions from D to D. Let A be an adversary with access to an oracle and that returns a bit. Then

$$\mathbf{Adv}^{\mathrm{prp}}_{F}(A) = \Pr\left[\, f \xleftarrow{\$} F \;:\; A^{f(\cdot)} = 1 \,\right] - \Pr\left[\, g \xleftarrow{\$} \mathsf{Perm}(D) \;:\; A^{g(\cdot)} = 1 \,\right]$$

denotes the PRP-advantage of A in distinguishing a random instance of F from a random permutation. Intuitively, we say that F is a secure PRP if the PRP-advantages of all adversaries using reasonable resources is small.

We often model block ciphers as pseudorandom functions or permutations. In this case, given a block cipher $E : \{0,1\}^k \times \{0,1\}^L \to \{0,1\}^L$, we use $E_K(\cdot)$, $K \in \{0,1\}^k$, to denote the function $E(K, \cdot)$ and we use $f \xleftarrow{\$} E$ as short hand for $K \xleftarrow{\$} \{0,1\}^k$; $f \leftarrow E_K$. We call k the key length of E and we call L the block length.

3 The CWC mode of operation

We now describe the CWC mode of operation for 128-bit block ciphers. (See Appendix C for a description of the general CWC construction.)

If BC denotes a block cipher with 128-bit blocks and kl-bit keys, and if tl is the desired tag length for CWC in bits, then let CWC-BC-tl denote the CWC mode of operation instantiated with BC using tag length tl. Throughout the remainder of this section, fix BC and tl and let CWC-BC-tl $=$ $(\mathcal{K}, \mathsf{CWC\text{-}ENC}, \mathsf{CWC\text{-}DEC})$.

We associate to CWC-BC-tl the following sets:

$$\begin{aligned}
\mathsf{MsgSp}_{\mathsf{CWC\text{-}BC\text{-}tl}} &= \{\, x \in (\{0,1\}^8)^* \;:\; |x| \leq \mathsf{MaxMsgLen} \,\} \\
\mathsf{AdSp}_{\mathsf{CWC\text{-}BC\text{-}tl}} &= \{\, x \in (\{0,1\}^8)^* \;:\; |x| \leq \mathsf{MaxAdLen} \,\} \\
\mathsf{KeySp}_{\mathsf{CWC\text{-}BC\text{-}tl}} &= \{0,1\}^{\mathsf{kl}} \\
\mathsf{NonceSp}_{\mathsf{CWC\text{-}BC\text{-}tl}} &= \{0,1\}^{88}
\end{aligned}$$

where MaxMsgLen and MaxAdLen are both $128 \cdot (2^{32} - 1)$. That is, the payload and associated data spaces for CWC-BC-tl consist of all strings of octets that are at most $2^{32} - 1$ blocks long.

3.1 The CWC core

The key generation algorithm $\mathcal{K}$ returns a randomly selected key from $\mathsf{KeySp}_{\mathsf{CWC\text{-}BC\text{-}tl}}$ (i.e., the key generation returns a random kl-bit string).

The encryption algorithm CWC-ENC works as follows:

Algorithm $\mathsf{CWC\text{-}ENC}_K(N, A, M)$ // CWC encryption
  $\sigma \leftarrow \mathsf{CWC\text{-}CTR}_K(N, M)$
  $\tau \leftarrow \mathsf{CWC\text{-}MAC}_K(N, A, \sigma)$
  Return $\sigma \| \tau$

where CWC-CTR and CWC-MAC are described in Section 3.2. The decryption algorithm CWC-DEC works as follows:

Algorithm $\mathsf{CWC\text{-}DEC}_K(N, A, C)$ // CWC decryption
  If $|C| < \mathsf{tl}$ then return INVALID
  Parse C as $\sigma \| \tau$ where $|\tau| = \mathsf{tl}$
  If $A \notin \mathsf{AdSp}_{\mathsf{CWC\text{-}BC\text{-}tl}}$ or $\sigma \notin \mathsf{MsgSp}_{\mathsf{CWC\text{-}BC\text{-}tl}}$ then return INVALID
  If $\tau \neq \mathsf{CWC\text{-}MAC}_K(N, A, \sigma)$ then return INVALID
  Return $\mathsf{CWC\text{-}CTR}_K(N, \sigma)$

3.2 The CWC subroutines

The remaining CWC algorithms are defined as follows:

Algorithm $\mathsf{CWC\text{-}CTR}_K(N, M)$ // CWC counter mode module
  $\alpha \leftarrow \lceil |M|/128 \rceil$
  For $i = 1$ to α do
    $\mathsf{ks}_i \leftarrow \mathsf{BC}_K(10^7 \| N \| \mathsf{tostr}(i, 32))$ // Note that 10^7 means a one bit followed by 7 zeros
  $\sigma \leftarrow$ (first $|M|$ bits of $\mathsf{ks}_1 \| \mathsf{ks}_2 \| \cdots \| \mathsf{ks}_\alpha) \oplus M$
  Return σ

Algorithm $\mathsf{CWC\text{-}MAC}_K(N, A, \sigma)$ // CWC authentication module
  $R \leftarrow \mathsf{BC}_K(\mathsf{CWC\text{-}HASH}_K(A, \sigma))$
  $\tau \leftarrow \mathsf{BC}_K(10^7 \| N \| 0^{32}) \oplus R$
  Return first tl bits of τ

Algorithm $\mathsf{CWC\text{-}HASH}_K(A, \sigma)$ // CWC universal hashing module
  $Z \leftarrow$ last 127 bits of $\mathsf{BC}_K(110^{126})$
  $K_h \leftarrow \mathsf{toint}(Z)$ // The same K_h value is used in every invocation of $\mathsf{CWC\text{-}HASH}_K$.
  $l \leftarrow$ minimum integer such that 96 divides $A \| 0^l$
  $l' \leftarrow$ minimum integer such that 96 divides $\sigma \| 0^{l'}$
  $X \leftarrow A \| 0^l \| \sigma \| 0^{l'}$; $\beta \leftarrow |X|/96$; $l_\sigma \leftarrow |\sigma|/8$; $l_A \leftarrow |A|/8$
  Break X into chunks $X_1, X_2, \ldots, X_\beta$ // $|X_1| = |X_2| = \cdots = |X_\beta| = 96$
  For $i = 1$ to β do
    $Y_i \leftarrow \mathsf{toint}(X_i)$
  $Y_{\beta+1} \leftarrow 2^{64} \cdot l_A + l_\sigma$ // Include the lengths of A and σ in the polynomial.
  $R \leftarrow Y_1 K_h^\beta + \cdots + Y_\beta K_h + Y_{\beta+1} \bmod 2^{127} - 1$
  Return $\mathsf{tostr}(R, 128)$ // Note: first bit of result will always be 0

3.3 Remarks

We now highlight some features of CWC, explain some of our design decisions, and discuss some of the alternatives we explored. We will have additional remarks in Section 4.3

Remark 3.1 [Computing the CWC-HASH polynomial.] The polynomial

$$Y_1 K_h^\beta + \cdots + Y_\beta K_h + Y_{\beta+1} \bmod 2^{127} - 1$$

in CWC-HASH can be computed using Horner's Rule as

$$((((Y_1)K_h + Y_2)K_h + \cdots)K_h + Y_\beta)K_h + Y_{\beta+1} \bmod 2^{127} - 1$$

or, if the values K_h^i are precomputed, computed directly. Furthermore, computation of the polynomial in CWC-HASH can be parallelized, as noted in the introduction, by splitting the polynomial into multiple polynomials in K_h^i for some i. Additional discussion appears in Section 5.

Remark 3.2 [On the size of the CWC-HASH coefficients.] All the coefficients $Y_1, \ldots, Y_\beta$ in CWC-HASH are 96-bit integers. It would easily be possible to define CWC to use coefficients up to 126-bits in size. Such an approach would speed up the computation by a ratio of 126/96 (nearly 4:3) when using Horner's rule, but would require considerable additional complexity to perform bit and byte shifting within the coefficients. The final $Y_{\beta+1}$ may, however, be larger than 96-bits since $Y_{\beta+1}$ does not have to be multiplied with anything.

Remark 3.3 [Why a single key.] It would be perfectly acceptable from a security perspective to make the block cipher key K and hash key K_h independent. The main motivation for using a single key, and deriving the hash key K_h from the block cipher key K, was simplicity of key management. From a performance perspective, we note that fetching key material can be a bottleneck in high-speed hardware.

Remark 3.4 [Separating block cipher inputs.] The input to the block cipher when generating the hash key K_h begins with the bits 11. All the inputs when generating CTR mode keystream begin with the bits 10. The input to the keystream generator in CWC-MAC has the last 32 bits all zero and the input to the block cipher in CWC-CTR never has the last 32 bits zero. All the outputs of CWC-HASH begin with a 0 bit. These properties ensure that there is never an overlap in the inputs between the different uses of the underlying block cipher. For example, the output of the universal hash function (which is enciphered with the block cipher) will never collide with one of the inputs to the block cipher in CWC-CTR. Essentially, separating the block cipher inputs in this way is what allows us to use a single block cipher key in all applications of the block cipher.

Remark 3.5 [Computing the universal hash subkey.] Although CWC-HASH shows the hash subkey K_h being computed upon every invocation, it is possible to compute K_h in the key generation step of CWC. Doing so would save one block cipher application per message but would require maintaining an additional 128 bits across invocations. We anticipate that in hardware, where fetching key material can be expensive, the hash subkey will be re-computed on every invocation of the encryption and decryption algorithms. In software, however, we anticipate that the subkey K_h will be computed once and maintained across invocations.

Remark 3.6 [On the choice of parameters.] The parameters (e.g., the nonce length and the way the nonce is encoded in the input to the block cipher) are fixed for CWC. This is in order to promote interoperability. In CWC the block counter length is set to 32 bits in order to allow CWC to be used with IPsec jumbograms and other large packets up to $2^{32}-1$ blocks long. The nonce length is set to 88 bits in order to handle future IPsec sequence numbers.

Remark 3.7 [Byte ordering.] CWC uses big-endian byte ordering. We do so for consistency purposes and in order to maintain compatibility with McGrew's ICM Internet-Draft [10] and the IETF, which strongly favors the big-endian byte-ordering.

Remark 3.8 [Handling arbitrary bit-length messages.] Although we could have specified CWC to take arbitrary bit-length messages as input (just change the definitions of the message spaces and compute $l_A \leftarrow |A|$ and $l_\sigma \leftarrow |\sigma|$ in CWC-HASH), we do not specify CWC this way simply because there does not appear to be a significant need to handle arbitrary bit-length messages and we do not consider it a good trade-off to define a mode for arbitrary bit-length messages at the expense of octet-oriented systems.

If, in the future, such a need arises, it will still be possible to modify the current CWC construction to take arbitrary bit-length messages as input without affecting interoperability with existing CWC implementations when octet-strings are communicated. Although other possibilities exist, one method would be to augment the computation of $Y_{\beta+1}$ in CWC-HASH as follows:

$$r_A \leftarrow |A| \bmod 8\,;\; r_\sigma \leftarrow |\sigma| \bmod 8\,;\; Y_{\beta+1} \leftarrow 2^{120} \cdot r_A + 2^{112} \cdot r_\sigma + 2^{64} \cdot l_A + l_\sigma\,.$$

Remark 3.9 [64-bit block ciphers.] It is possible to instantiate the general CWC paradigm (Appendix C) with 64-bit block ciphers like DES and 3DES. We do not do so in this paper since we are targeting future high-speed cryptographic applications.

Remark 3.10 [Initial counter for CTR-mode.] Motivated by EAX2 [2], one possible alternative to CWC might be to use $\mathsf{BC}_K(1110^5\|N)$ both as the value to encrypt R in $\mathsf{CWC\text{-}MAC}$ and as the initial counter to CTR mode-encrypt M (with the first two bits of the counter always set to 10). Other EAX2-motivated constructions also exist. For example, the tag might be set to $\mathsf{BC}_K(h(X_0\|N)) \oplus \mathsf{BC}_K(h(X_1\|A)) \oplus \mathsf{BC}_K(h(X_2\|\sigma))$, where X_0, X_1, X_2 are strings, none of which is a prefix of the other, and h is a parallelizable universal hash function, like $\mathsf{CWC\text{-}HASH}$ but hashing only single strings (as opposed to pairs of strings). Compared to CWC, these alternatives have the ability to take longer nonces as input, and, from a functional perspective, can be applied to strings up to 2^{126} blocks long. But we do not view this as a reason to prefer these alternatives over CWC. From a practical perspective, we do not foresee applications ever needing nonces longer than 11 octets, or needing to encrypt messages longer than $2^{32} - 1$ blocks. Moreover, from a security perspective, applications should not encrypt too many packets between rekeyings, implying that even 11 octet nonces are more than sufficient.

4 Theorem statements

The CWC mode is a provably secure AEAD scheme assuming that the underlying block cipher (e.g., AES) is a secure pseudorandom function or pseudorandom permutation. This is a quite reasonable assumption since most modern block ciphers (including AES) are believed to be pseudorandom. Furthermore, all provably-secure block cipher modes of operation that we are aware of make the same assumptions we make (and some modes, e.g. OCB [15], make even stronger, albeit still reasonable, assumptions).

The specific results for CWC appear as Theorem 4.1 and Theorem 4.2 below. In Appendix C we also present results for the general CWC paradigm, from which Theorems 4.1 and 4.2 follow.

4.1 Integrity/authenticity

Theorem 4.1 [Integrity/authenticity of CWC.] Let $\mathsf{CWC\text{-}BC\text{-}tl}$ be as in Section 3. (Recall that the block cipher is BC and tag length is tl.) Consider a nonce-respecting AUTH adversary A against $\mathsf{CWC\text{-}BC\text{-}tl}$. Assume the execution environment allows A to query its oracle with associated data that are at most $n \le \mathsf{MaxAdLen}$ bits long and with messages that are at most $m \le \mathsf{MaxMsgLen}$ bits long. Assume A makes at most $q-1$ oracle queries and the total length of all the payload data (both in these $q-1$ oracle queries and the forgery attempt) is at most μ. Then given A we can construct a PRF adversary B_A and a PRP adversary C_A against BC such that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \le \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{BC}}(B_A) + \frac{n+m}{2^{133}} + \frac{1}{2^{125}} + \frac{1}{2^{\mathsf{tl}}}$$

and

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \le \mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} + \frac{n+m}{2^{133}} + \frac{1}{2^{125}} + \frac{1}{2^{\mathsf{tl}}} \,. \tag{1}$$

Furthermore, the experiments for B_A and C_A take the same time as the experiment for A and B_A and C_A make at most $\mu/128 + 3q + 1$ oracle queries. ▮

The above theorem means that if the underlying block cipher is a secure pseudorandom function or a secure pseudorandom permutation, then $\mathsf{CWC\text{-}BC}$ will preserve authenticity. If the underlying block cipher is something like AES, then this initial assumption seems quite reasonable and, therefore, $\mathsf{CWC\text{-}AES}$ will preserve authenticity.

Let us elaborate on this reasoning. Assume BC is a secure block cipher. This means that $\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C)$ must be small for all adversaries C using reasonable reasonable resources and, in particular, this means that, for C_A as described in the above theorem statement, $\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A)$ must be small assuming that A uses reasonable resources. And if $\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A)$ is small and μ, q, m and n are small, then, because of the above equations, $\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A)$ must also be small as well. I.e., any adversary A using reasonable resources will only be able to break the authenticity of CWC-BC-tl with some small probability.

Let us consider some concrete examples. Let $n = \mathsf{MaxAdLen}$ and $m = \mathsf{MaxMsgLen}$, which is the maximum possible allowed by the CWC-BC construction. Then Equation 1 becomes

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \leq \mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} + \frac{1}{2^{93}} + \frac{1}{2^{\mathsf{tl}}} \,.$$

If we limit the number of applications of CWC-BC between rekeyings to some reasonable value such as $q = 2^{32}$, if we limit the total number of payload bits between rekeyings to $\mu = 2^{50}$, and if we take $\mathsf{tl} \geq 43$, then the above equation becomes

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \leq \mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{1}{2^{41}}$$

which means that, assuming that the underlying block cipher is a secure PRP, an attacker will not be able to break the unforgeability of CWC-BC-tl with probability much greater than 2^{-41}.

4.2 Privacy

Theorem 4.2 [Privacy of CWC.] Let CWC-BC-tl be as in Section 3. Then given a nonce-respecting IND\$-CPA adversary A against CWC-BC-tl one can construct a PRF adversary B_A and a PRP adversary C_A against BC such that

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{BC}}(B_A)$$

and, if A makes at most q oracle queries totaling at most μ bits of payload message data,

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \leq \mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} \,. \tag{2}$$

Furthermore, the experiments for B_A and C_A take the same time as the experiment for A and B_A and C_A make at most $\mu/128 + 3q + 1$ oracle queries. ▮

We interpret Theorem 4.2 in the same way we interpreted Theorem 4.1. In particular, this theorem shows that if BC is a secure pseudorandom function or pseudorandom permutation, then CWC-BC-tl preserves privacy under chosen-plaintext attacks.

As a concrete example let us again take $q = 2^{32}$ and $\mu = 2^{50}$. Then Equation 2 becomes

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A) \leq \mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{1}{2^{42}}$$

which means that, assuming that the underlying block cipher is a secure PRP, an attacker will not be able to break the privacy of CWC-BC-tl with advantage much greater than 2^{-42}.

CHOSEN-CIPHERTEXT PRIVACY. Since CWC-BC-tl preserves privacy under chosen-plaintext attacks (Theorem 4.2) *and* provides integrity (Theorem 4.1) assuming that BC is pseudorandom, it also provides privacy under chosen-ciphertext attacks. (See [1, 14] for a discussion of the relationship between chosen-plaintext privacy, integrity, and chosen-ciphertext privacy; this relationship was also used, for example, by [15].)

4.3 Remarks

Remark 4.3 [On the length of the hash subkey.] It is possible to use smaller hash subkeys K_h in $\mathsf{CWC\text{-}HASH}$ (simply truncate $\mathsf{BC}_K(110^{126})$ appropriately). Let hkl denote the length of the hash subkey in an altered construction. If $\mathsf{hkl} < 127$, then the upper-bound in Equation 1 becomes

$$\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} + \frac{(n+m)/96 + 2}{2^{\mathsf{hkl}}} + \frac{1}{2^{\mathsf{tl}}} .$$

Consider an application that sets hkl to 96. If we replace m and n by their maximum possible values, the upper-bound becomes

$$\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} + \frac{1}{2^{62}} + \frac{1}{2^{\mathsf{tl}}} .$$

Since 2^{-62} is already very small (and, in fact, dominated by the $(\mu/128 + 3q + 1)^2 \cdot 2^{-129}$ term for some reasonable values of q and μ), from a provable-security perspective, developers would be justified in using 96-bit hash subkeys.

Rather than use shorter hash subkeys, however, our current CWC instantiation in Section 3 uses 127-bit hash subkeys. We do so for several reasons. First, in hardware, to obtain maximum speed, one would parallelize the CWC hash function by evaluating, for example, two polynomials in K_h^2 in parallel. Since K_h^2 would generally not be 96-bits long, there is no performance advantage with using 96-bit subkeys K_h in this situation. In software, the use of 96-bit hash subkeys could lead to improved performance when evaluating the polynomial using Horner's rule. However, the performance of such a construction is essentially equivalent to the performance of the current construct when not using Horner's rule but using pre-computed powers of K_h. Since we believe that high-performance implementations will not benefit from the use of 96-bit hash subkeys (i.e., the additional 31 key bits come with no or negligible additional cost), we have chosen to fix the length of our hash subkeys to 127 bits.

Developers of CWC derivatives may, however, wish to use shorter hash subkeys, and we do not prevent that (although we do suggest referring to such modes in such a way as to avoid confusion with $\mathsf{CWC\text{-}BC}$). We also suggest that developer's understand the impact of using shorter hash subkeys. For example, using a 64-bit hash subkey would increase the upper-bound on the probability of an adversary forging to around 2^{-30}, which may be too large for some applications.

Remark 4.4 [On computing the tag.] In CWC the MAC consisted of hashing (A, σ), enciphering the hash with the block cipher, and then XORing the result with some keystream (i.e., in the current proposal the tag is $\mathsf{BC}_K(10^7 \| N \| 0^{32}) \oplus \mathsf{BC}_K(\mathsf{CWC\text{-}HASH}_K(A, \sigma)))$. One question the reader might have is whether two block cipher invocations are necessary. We first comment that the cost of two block cipher operations per MAC is not particularly significant compared to the total cost of CWC. $\mathsf{CWC\text{-}AES}$ as currently specified already achieves its design goal of encrypting 10 Gbps in hardware. And, in software, the extra cost of one block cipher operation is quite minor for average packets, and less than approximately 15% for 64-byte packets. Nevertheless, the use of two block cipher applications for the tag might seem aesthetically unappealing to some.

Instead of the two block cipher applications, one could use $\mathsf{BC}_K(h'_K(N, A, \sigma))$ as the tag, where h' is a modified version of $\mathsf{CWC\text{-}HASH}$ designed to hash 3-tuples instead of pairs of strings (this is important because the nonce must also be authenticated). The main disadvantage of this approach is that it would change the upper-bound in Equation 1 to

$$\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 3q + 1)^2}{2^{129}} + q^2 \cdot \left(\frac{n+m}{2^{133}} + \frac{1}{2^{125}} \right) + \frac{1}{2^{\mathsf{tl}}}$$

(note the new q^2 term). If we set $n = \mathsf{MaxAdLen}$, $m = \mathsf{MaxMsgLen}$, $q = 2^{32}$, and $\mu = 2^{50}$, then for any $\mathsf{tl} \geq 29$, we get that the advantage of an adversary in breaking the unforgeability of this modified CWC variant is upper-bounded by 2^{-27}, which, although not extremely large, is worse than the upper-bound of 2^{-41} we get using Equation 1. Even if n and m are at most one million blocks long, we see that the integrity upper-bound for the altered CWC construction is worse than the upper-bound for the CWC construction we present in Section 3. More generally, this means that for reasonable values of n, m, q, μ, the insecurity upper-bounds of this alternative will be worse than the insecurity upper-bounds of the CWC mode described in Section 3. Furthermore, the upper-bound would be even worse if one keys the hash function with shorter keys, which some developers might choose to do (recall Remark 4.3).

Another possible way to reduce the number of block cipher invocations necessary to compute the MAC would be to take the output of the current hash function and run it through another hash function that is almost-XOR-universal (see Appendix C for a description of this property). However, this approach is not attractive because it requires additional key material. In particular, while this approach may save one block cipher operation, in hardware the block cipher operation is actually smaller and simpler than managing the extra key material, given that the hardware already has a block cipher encryptor running at high speed.

Another possibility would be to use something like $\mathsf{BC}_K(N) + Y_1 K_h^{\beta+2} + \cdots + Y_\beta K_h^3 + l_A K_h^2 + l_\sigma K_h \bmod 2^{127} - 1$, encoded as a 127-bit string and truncated to tl bits, as the MAC (here $\mathsf{BC}_K(N)$ is interpreted as an integer). Doing so would, however, result in a new integrity upper-bound

$$\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 2q + 1)^2 + 4q + 4}{2^{129}} + \frac{(n+m)/96 + 5}{2^{\mathsf{tl}}} \; .$$

If we take n and m to be $\mathsf{MaxAdLen}$ and $\mathsf{MaxMsgLen}$, respectively, then the upper-bound becomes

$$\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{BC}}(C_A) + \frac{(\mu/128 + 2q + 1)^2 + 4q + 4}{2^{129}} + \frac{2^{34}}{2^{\mathsf{tl}}} \; .$$

Compared to Equation 1, we see the presence of a $2^{34-\mathsf{tl}}$ term. This means that, in some situations, when using the above upper-bound as a guide for parameter selection, tag lengths must be longer than one might expect. For example, if $\mathsf{tl} = 32$, then the above equation would upper-bound the advantage of an adversary against this modified construction as 1. This means that 32-bit tags should not be used with this modified construction when authenticating long messages. While one might consider this more of a "certificational" problem than a real problem, we view this property as undesirable. Hence our decision to specify CWC as in Section 3.

5 Performance

5.1 Hardware

Since one of our main goals is to achieve 10 Gbps in hardware, let us focus first on hardware costs. As noted in the introduction, using 0.13 micron CMOS ASIC technology, it should take approximately 300 Kgates to achieve 10 Gbps throughput for CWC-AES. This estimate, which is applicable to AES with all key lengths, includes four AES counter-mode encryption engines, each running at 200 MHz and requiring about 25Kgates each. In addition, there are two 32x128-bit multiply/accumulate engines, each running at 200 MHz with a latency of four clocks, one each for the even and odd polynomial coefficients. Of course, simply keeping these engines "fed" may be quite a feat in itself, but that is generally true of any 10 Gbps path. Also, there may well be better methods to structure an implementation, depending on the particular ASIC vendor library and technology, but, regardless of the implementation strategy, 10 Gbps is quite achievable because of the inherent parallelism of CWC.

Since OCB is CWC's main competitor for high-speed environments, it is worth comparing CWC with OCB instantiated with AES (we do not compare CWC with CCM and EAX here since the latter two are not parallelizable). We first note that CWC-AES saves some gates because we only have to implement AES encryption in hardware. However, at 10 Gbps, OCB still probably requires only about half the silicon area of CWC-AES. The main question for many hardware designers is thus whether the extra silicon area for CWC-AES costs more than three royalty payments, as well as negotiation costs and overhead. Our estimates indicate that, given today's silicon costs, the extra silicon for CWC-AES is probably cheaper than the IP fees for OCB.

5.2 Software

CWC-AES can also be implemented efficiently in software. The amortized (per-byte) cost for the actual encryption portion is simply the cost of the counter mode encryption, which itself is just the cost of AES, plus the overhead of managing a counter. The counter may be implemented as an increment on a 32-bit integer. On some platforms, this may require two byte swaps as well. Assuming the AES implementation takes its plaintext from memory instead of registers (as will generally be the case on x86 platforms), there may also be additional overhead in storing the counter once per 16-octet block. Of course, there is also a per-message fixed cost that primarily involves loading the nonce. Many systems will divide the nonce into a random, per-connection salt and a message counter. If the message counter is 32 bits, then the additional per-message overhead should be as low as could reasonably be expected.

The performance of the polynomial hash function is more complicated to analyze. The first available implementations, which have all been C-based, work at speeds ranging from 18.6 cpb (Brian Gladman's implementation) to 33 cpb (the reference implementation) on a Pentium III, depending on the implementation strategy. There is plenty of room for improvement, as even Gladman's implementation could be made significantly faster by moving to assembly and adopting precomputation tricks first used in this type of hash by Bernstein [3]. We discuss these tricks shortly.

Comparing existing CWC-HASH and CBC-MAC implementations. Since both CCM and EAX use CBC-MAC, it is interesting and useful to contrast the speed of CWC-HASH to the speed of CBC-MAC using AES-128, tested under the same operating environment. The best reported AES implementation is Helger Lipmaa's commercial implementation, which is implemented in assembler and runs CBC-MAC in as few as 14.5 cycles per byte on a Pentium III. However, it's unclear how reflective of real-world operating conditions that number is. In fact, we've not been able to approach these speeds in freely available implementations for a Unix-based platform, even those that are supposed to run a mere 33% slower. Using the assembly-based implementation from OpenSSL (which is not advertised as significantly speedy), we were able to achieve 44.8 cpb in the same environment in which we tested CWC-AES. When running OpenSSL's own speed tests, the results were actually a few cycles slower per byte.

There are freely available AES implementations that are reported to go significantly faster. For example, Brian Gladman's assembly-based version is said to run CBC-MAC at about 17.6 Pentium III cpb, and the C version is said to run at 22.8 cpb.[5] The asm version we could not use

[5] To obtain these numbers, we started with the numbers at `http://www.tcs.hut.fi/~helger/aes/rijndael.html` and then added the minimal number of cycles for the additional overhead for CBC-MAC (e.g., the XORs).

directly since it is designed for Microsoft assemblers. And, on several varieties of UNIX, neither Gladman's C-based version nor any other we could find ran anywhere near the 23 cpb range (in fact, Gladman's version was at least as slow as the OpenSSL version). We note that widely reported timings for Gladman's C implementation were calculated using Microsoft's compiler and platform. The difference is probably largely explained by differences in the two platforms, including the quality of the optimizer, the calling conventions used, and so on.

All in all, we see evidence that CBC-MAC implemented using an assembly-based, highly optimized AES implementation may be able to run about as fast as Gladman's C implementation of CWC-HASH, which itself is still largely unoptimized. Additionally, we note that there seems to be significant practical difficulties in achieving speeds comparable to CWC-HASH when using CBC-MAC with AES. We shall discuss approaches to further improve the performance of the CWC-HASH algorithm shortly.

Implementation strategies. There are two general approaches to implementing CWC-HASH in software. The first is factoring using Horner's rule. Second, one can evaluate the polynomial without factoring, which may seem more expensive, but can actually be faster if one precomputes powers of the hash key at setup time. The advantage with precomputation was first observed by Bernstein in the context of his hash127 [3].

Assuming that one precomputes enough that no messages will ever require dynamic key computation, then there are the same number of 96x127-bit multiply operations and the same number of addition operations in the two strategies. Otherwise, if one has enough precomputed material to process n blocks of data, the cost can be held to *one* additional 127x127-bit modular multiplication for *each* n blocks of data processed. For a message that is a multiple of n blocks in length, one processes the first n blocks, then multiplies the intermediate result by K_h^n. After that, one can continue processing data with the same precomputed values. If the total message is m blocks long and m is known in advance, one could *first* process $m \bmod n$ blocks, multiply by a K_h^n, then process in n-block chunks as before. Alternately, as long as the end of the message is known n-blocks in advance, one could process n-block chunks, and then finishing off the final $m \bmod n$ block using Horner's rule. Or, if m is not known in advance, one could process the message in n-block chunks and then multiply by a precomputed power of K_h^{-1} once the end of the message hash been reached.

The precomputation approach has the drawback that the message must be buffered in parts. The larger the buffering, the more precomputation can be done. Naturally, precomputation requires extra memory, but that is usually cheap and plentiful in a software-based environment.

There is an advantage to precomputation. With Horner's rule, we must build a 127-bit by 127-bit multiplier. With the precomputation approach, assuming we precompute enough values of the hash subkey, we just multiply 96-bit coefficients with a 127-bit expanded key. Indeed, a straightforward C version of the precomputation approach runs in just under 24 cpb on a Pentium III, which is better than 25% faster than our straightforward implementation using Horner's rule.

Using 32-bit multiplies, the precomputation approach requires 12 32-bit multiplies per 96 bits of (padded) plaintext, as well as 17 adds, all of which may carry. In assembly, most of these carry operations can be implemented for free, or close to it by using a special variant of the add instruction that adds in the operand as well as the value of the carry from the previous add operation. But when implemented in C, they will generally compile to code that requires a conditional branch and an extra addition. An implementation using Horner's rule requires an additional four multiples and three additions with carry per block, adding about 33% overhead, since the multiplies dominate the additions.

A 64-bit platform only requires four multiplies and four adds (which may all carry), no matter the implementation strategy taken. The multiply being far more expensive than other operations, we would thus expect a 64-bit integer implementation to run in one third the time of a 32-bit implementation, assuming that the cost of primitive operations does not increase.

Exploiting the parallelism of some instruction sets. On most platforms, it turns out, the integer execution unit is not the fastest way to implement CWC-HASH. Many platforms have multimedia instructions that can be used to speed up the implementation. As another alternative, Bernstein demonstrated that, on most platforms, the floating point unit can be used to implement this class of hash far more efficiently than can be done in the integer unit. This is particularly true on the x86 platform, where, in contrast to using the standard registers, two floating point multiples can be started in close proximity without introducing a pipeline stall. That is, the x86 can effectively perform two floating-point operations in parallel.

The disadvantage of using floating-point registers is that the operands for the individual multiplies need to be small, so that the operations can be done without loss of precision. On the x86, Bernstein multiplies 24-bit values, allowing the sums of product terms to fit into double precision values with 53 bits of precision, without loss of information. Bernstein details many ways to optimize this sort of calculation in [3].

Bernstein's hash127 is a polynomial hash using the same modulus as CWC-HASH. In fact, there are only two differences between the cores of Bernstein's hash127 and CWC-HASH. The first is that Bernstein uses signed coefficients, whereas CWC-HASH uses unsigned coefficients. We do not see where this decision would have an impact on efficiency. The other difference is that Bernstein uses 32-bit coefficients, whereas CWC-HASH uses 96-bit coefficients. While both solutions average one multiplication per byte when using integer math, Bernstein's solution requires only .75 additions per byte, whereas CWC-HASH requires 1.42 additions per byte, nearly twice as many.

Using 32-bit multiplies to build a 96x127 multiplier (assuming precomputation), CWC-HASH should therefore perform no worse than at half the speed of hash127. When using 24-bit floating point coefficients to build a multiply (without applying any non-obvious optimizations), hash127 requires 12 multiplies and 16 adds per 32-bit word. CWC can get by with 8 multiples per word and 12.67 additions per word. This is because a 96-bit coefficient fits exactly into four 24-bit values, meaning we can use a 6x4 multiply for every three words. With 32-bit coefficients, we need to use two 24-bit values to represent each coefficient, resulting in a single 6x2 multiply that needs to be performed for each word.

In an implementation that factors using Horner's rule, instead of precomputing, then both hash127 and CWC-HASH would need full 127x127 multiplies for every coefficient, which would make CWC-HASH significantly more desirable, since it uses one third fewer coefficients per message. (As noted in Remark 3.2, the fact that parallel high-speed hardware implementations will likely use Horner's rule is one reason we chose to use 96-bit coefficients.)

Brian Gladman's implementation of CWC-HASH uses floating point arithmetic, and factors the polynomial using Horner's rule, instead of performing precomputation to achieve extra speed. All in all, nothing about the CWC hash indicates that it should run any worse than half the speed of hash127, if implemented in a similar manner using the floating point registers and precomputation. That upper-bound paints an encouraging picture for CWC performance, because hash127 on the same Pentium III used above runs in 3.8 cpb when implemented in assembly, leveraging the floating point registers and precomputation. This indicates that a well-optimized software version of CWC-HASH should run no slower than 8 cycles per byte, which should be more than acceptable for all practical applications of CWC.

It may be possible to improve on Bernstein's techniques. For example, literature from the gaming community [5] indicates that one can use both integer and floating point registers in parallel. One might be able to use the integer registers to operate on every few blocks, and expect to see a speedup.

5.3 Performance summary

CWC-AES is efficient and parallelizable, where it can achieve 10 Gbps in hardware using a single hardware engine. We view this as one of CWC's primary advantages over competitor patent-free modes CCM and EAX since CCM and EAX are based on CBC-MAC and therefore cannot achieve such speeds using a single hardware engine.

CWC-AES is also efficient in software. In software, the time for a CWC-AES computation is split between the time spent doing the counter mode encryption and the time computing the CWC-HASH. The time for the counter mode portion of CWC-AES is essentially the time for any counter mode implementation. A naive implementation of CWC-HASH runs at 33 cpb on a Pentium III. The best implementation we have tested to date, due to Gladman, runs at 18.6 cpb on a Pentium III, which is better than the speeds we have measured for CBC-MAC implementations using existing crypto libraries. Using the techniques from Bernstein [3], we anticipate speeds of 8 cpb or better for CWC-HASH.

We do not claim that CWC-AES will be particularly efficient on low-end CPUs such as 8-bit smartcards. However, our goal was not to develop an efficient AEAD scheme for such low-end CPUs. Rather, our goal was to develop a parallelizable and efficient AEAD scheme for high-end systems and 10 Gbps hardware.

6 Conclusions

In this work we present CWC, a new block cipher-based AEAD scheme. To the best of our knowledge, CWC is currently the only dedicated block cipher-based AEAD scheme that is simultaneously (1) parallelizable and efficient, (2) provably-secure, and (3) patent-free. We believe that simultaneously having these properties makes CWC a strong candidate for use in future high-performance cryptographic applications.

Acknowledgments

We would like to thank Peter Gutmann, David McGrew, and David Wagner for their comments. Additionally, we would like to thank Brian Gladman for helping to validate our test vectors and for providing us timing information for an optimized CWC implementation. T. Kohno was supported by a National Defense Science and Engineering Fellowship.

References

[1] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer-Verlag, Berlin Germany, Dec. 2000.

[2] M. Bellare, P. Rogaway, and D. Wagner. A conventional authenticated-encryption mode, Apr. 2003. Available at `http://eprint.iacr.org/2003/069/`.

[3] D. Bernstein. Floating-point arithmetic and message authentication, 2000. Available at `http://cr.yp.to/papers.html#hash127`.

[4] V. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In *Fast Software Encryption 2001*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 2001.

[5] C. Hecker. Perspective texture mapping, part v: It's about time. *Game Developer*, Apr. 1996. Available at `http://www.d6.com/users/checker/pdfs/gdmtex5.pdf`.

[6] C. Jutla. Encryption modes with almost free message integrity. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 529–544. Springer-Verlag, Berlin Germany, May 2001.

[7] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In B. Schneier, editor, *Fast Software Encryption 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, Berlin Germany, Apr. 2000.

[8] H. Krawczyk. LFSR-based hashing and authentication. In Y. Desmedt, editor, *Advances in Cryptology – CRYPTO '94*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, Aug. 1994.

[9] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer-Verlag, Berlin Germany, Aug. 2001.

[10] D. McGrew. Integer counter mode, Oct. 2002. Available at `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-icm-00.txt`.

[11] D. McGrew. The truncated multi-modular hash function (TMMH), version two, Oct. 2002. Available at `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-tmmh-00.txt`.

[12] D. McGrew. The universal security transform, Oct. 2002. Available at `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-ust-00.txt`.

[13] P. Rogaway. Bucket hashing and its applications to fast message authentication. *Journal of Cryptology*, 12:91–115, 1999.

[14] P. Rogaway. Authenticated encryption with associated data. In *Proceedings of the 9th Conference on Computer and Communications Security*, Nov. 2002.

[15] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proceedings of the 8th Conference on Computer and Communications Security*, pages 196–205. ACM Press, 2001.

[16] P. Rogaway and D. Wagner. A critique of CCM, Apr. 2003. Available at `http://eprint.iacr.org/2003/070/`.

[17] D. Stinson. Universal hashing and authentication codes. *Designs, Codes and Cryptography*, 4:369–380, 1994.

[18] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

[19] D. Whiting, N. Ferguson, and R. Housley. Counter with CBC-MAC (CCM). Submission to NIST. Available at `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/`, 2002.

A Summary of properties

In this appendix we summarize some of the properties of CWC. We include all of the properties listed in the submission guidelines on the NIST Modes of Operation website. We also discuss some additional properties that we feel are important.

Security function. CWC is a provably secure *authenticated encryption* with associated data (AEAD) scheme. Informally, this means that the encapsulation algorithm, on input a pair of messages (A, M) and some nonce N, encapsulates (A, M) in a way that protects the privacy of M and the integrity of both A and M. Our formal security statements appear in Section 4 and the proofs appear in Appendix C.

Error propagation. Assuming that the underlying block cipher is a secure pseudorandom function or permutation, any attempt, by an adversary using reasonable resources, to forge a new ciphertext will, with very high probably, be detected. This follows from the fact that CWC is a provably-secure AEAD scheme.

Synchronization. Synchronization is based on the nonce. As with other nonce-based AEAD schemes, the nonce must either be sent with the ciphertext or the receiver must know how to derive the nonce on its own.

Parallelizability. CWC is *parallelizable*. The amount of parallelism for the hashing portion can be determined by the implementor without affecting interoperability.

Keying material required. CWC is defined to be a *single-key* AEAD scheme. However, CWC does internally use two keys (the main block cipher key and a hash key which is derived using the block cipher key). Implementors can decide whether to store the derived hash key in memory or whether to re-derive it as needed.

Counter/IV/nonce requirements. CWC uses a 11-octet *nonce*. CWC is provably secure as long as one does not query the encryption algorithm twice with the same nonce. Although it is possible to instantiate the generic CWC paradigm with other nonce lengths, for CWC the nonce size is fixed at 11-octets in order to minimize interoperability issues.

Memory requirements. The software memory requirements are basically those of the underlying block cipher. For example, fast AES in software requires 4K bytes of table, and about 200 bytes of expanded key material. In some situations, software implementations may precompute powers of the hash subkey.

Pre-processing capability. The underlying CTR mode keystream can be *precomputed*. The only block cipher input that cannot be precomputed is the output of CWC-HASH.

CWC can preprocess its associated data, thereby reducing computation time if the associated data remains static or changes only infrequently.

Message length requirements. The associated data and message can both be any string of octets with length at most $128 \cdot (2^{32} - 1)$ bits. Because there does not appear to be a need to handle strings of arbitrary bit-length, CWC as currently specified cannot encapsulate arbitrary bit-length messages. (As discussed in Section 3, it is easy to modify CWC to handle arbitrary bit-length messages, if desired.)

Ciphertext expansion. The ciphertext expansion is the minimum possible while still providing a tl-bit tag. That is, on input a pair (A, M), a nonce N, and a key K, $\mathsf{CWC\text{-}ENC}_K(N, A, M)$ outputs a ciphertext C with length $|C| = |M| + \mathsf{tl}$.

Block cipher invocations. If the hash subkey K_h is computed as part of the key generation process and not during each invocation of the CWC encapsulation routine, then CWC makes one block cipher invocation during key setup and $\lceil |M|/128 \rceil + 2$ block cipher invocations during encapsulation and decapsulation. If the hash subkey K_h is not computed as part of the key generation process, then CWC makes no block cipher invocations during key setup and $\lceil |M|/128 \rceil + 3$ block cipher invocations during encapsulation and decapsulation.

Provable security. CWC is a *provably-secure* AEAD scheme assuming that the underlying block cipher (e.g., AES) is a secure pseudorandom function or permutation. The proofs of security do not require the block cipher to satisfy the strong notion of super-pseudorandomness required by some other block cipher modes of operation.

Number of options and interoperability. CWC uses a minimal number of options. The only options are the choice of the underlying block cipher and the tag length. Having fewer options makes interoperability easier.

On-line. The CWC encryption algorithm is on-line. This means that CWC can process data as it arrives, rather than waiting for the entire message to be buffered before beginning the encryption processes. This may be advantageous when encrypting streaming data sources. (Note, however, that, like any other AEAD scheme, the decryptor should still buffer the entire message and check the tag τ before revealing the plaintext and associated data.)

Patent status. To the best of our knowledge CWC is *not* covered by any patents.

Performance. CWC is efficient in both hardware and software. In hardware, CWC can process data at 10 Gbps.

Simplicity. Although simplicity is a matter of perspective, we believe that CWC is a very simple construction. It combines standard CTR mode encryption with the evaluation of a polynomial modulo $2^{127}-1$. Because of its simplicity, we believe that CWC is easy to implement and understand.

B Intellectual property statement

The authors hereby explicitly release any intellectual property rights to CWC mode into the public domain. Further, the authors are not aware of any patent or patent application anywhere in the world that cover this mode.

C Proofs of security

Before proving Theorem 4.1 and Theorem 4.2, we first state results about the general CWC paradigm (see Lemma C.6 and Lemma C.7 below). We then show how Theorems 4.1 and 4.2 follow from Lemmas C.6 and C.7. We then prove these two lemmas.

C.1 More definitions

We begin with a few additional definitions.

Universal hash functions. A hash function $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ consists of two algorithms and is defined over some key space $\mathsf{KeySp}_{\mathcal{HF}}$, some message space $\mathsf{MsgSp}_{\mathcal{HF}}$, and some hash space $\mathsf{HashSp}_{\mathcal{HF}}$. The randomized key generation algorithm returns a random key $K \in \mathsf{KeySp}_{\mathcal{HF}}$; we denote this as $K \stackrel{\$}{\leftarrow} \mathcal{K}_h$. The deterministic hash algorithm takes a key $K \in \mathsf{KeySp}_{\mathcal{HF}}$ and a message $M \in \mathsf{MsgSp}_{\mathcal{HF}}$ and returns a hash value $h \in \mathsf{HashSp}_{\mathcal{HF}}$; we denote this as $h \leftarrow \mathcal{H}_K(M)$. Let $H \stackrel{\$}{\leftarrow} \mathcal{HF}$ be shorthand for $K \stackrel{\$}{\leftarrow} \mathcal{K}_h$; $H \leftarrow \mathcal{H}_K$.

The hash function $\mathcal{HF}$ is said to be ϵ-*almost universal* (ϵ-AU) if for all distinct $m, m' \in \mathsf{MsgSp}_{\mathcal{HF}}$,

$$\Pr\left[\, H \xleftarrow{\$} \mathcal{HF} \;:\; H(m) = H(m') \,\right] \leq \epsilon \;.$$

The hash function $\mathcal{HF}$ is said to be ϵ-*almost* XOR *universal* (ϵ-AXU) if $\mathsf{HashSp}_{\mathcal{HF}} = \{0,1\}^n$ for some positive integer n and for all distinct $m, m' \in \mathsf{MsgSp}_{\mathcal{HF}}$ and $c \in \{0,1\}^n$,

$$\Pr\left[\, H \xleftarrow{\$} \mathcal{HF} \;:\; H(m) \oplus H(m') = c \,\right] \leq \epsilon \;.$$

MESSAGE AUTHENTICATION. A nonced message authentication scheme $\mathcal{MA} = (\mathcal{K}_m, \mathcal{T}, \mathcal{V})$ consists of three algorithms and is defined over some key space $\mathsf{KeySp}_{\mathcal{MA}}$, some nonce space $\mathsf{NonceSp}_{\mathcal{MA}}$, some message space $\mathsf{MsgSp}_{\mathcal{MA}}$, and some tag space $\mathsf{TagSp}_{\mathcal{MA}}$. The randomized key generation algorithm returns a key $K \in \mathsf{KeySp}_{\mathcal{MA}}$; we denote this as $K \xleftarrow{\$} \mathcal{K}_m$. The deterministic tagging algorithm $\mathcal{T}$ takes a key $K \in \mathsf{KeySp}_{\mathcal{MA}}$, a nonce $N \in \mathsf{NonceSp}_{\mathcal{MA}}$, and a message $M \in \mathsf{MsgSp}_{\mathcal{MA}}$ and returns a tag $\tau \in \mathsf{TagSp}_{\mathcal{MA}}$; we denote this process as $\tau \leftarrow \mathcal{T}_K^N(M)$ or $\tau \leftarrow \mathcal{T}_K(N, M)$. The deterministic verification algorithm $\mathcal{V}$ takes as input a key $K \in \mathsf{KeySp}_{\mathcal{MA}}$, a nonce $N \in \mathsf{NonceSp}_{\mathcal{MA}}$, a message $M \in \mathsf{MsgSp}_{\mathcal{MA}}$, and a candidate tag $\tau \in \{0,1\}^*$, computes $\tau' = \mathcal{T}_K^N(M)$, and returns accept if $\tau' = \tau$ and returns reject otherwise.

Let F be a forging adversary and consider an experiment in which we first pick a random key $K \xleftarrow{\$} \mathcal{K}_m$ and then run F with oracle access to $\mathcal{T}_K(\cdot,\cdot)$. We say that F *forges* if F returns a triple (N, M, τ) such that $\mathcal{V}_K^N(M, \tau) = \mathsf{accept}$ but F did not make a query (N, M) to $\mathcal{T}_K(\cdot,\cdot)$ that resulted in a response τ. Then

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathcal{MA}}(F) = \Pr\left[\, K \xleftarrow{\$} \mathcal{K}_m \;:\; F^{\mathcal{T}_K(\cdot,\cdot)} \text{ forges } \,\right]$$

denotes the UF-*advantage* of F in breaking the *unforgeability* of $\mathcal{MA}$. An adversary is *nonce-respecting* if it never queries its tagging oracle with the same nonce twice. Intuitively, $\mathcal{MA}$ is unforgeable if the UF-advantage of all nonce-respecting adversaries with reasonable resources is small.

C.2 The general CWC construction

We now describe our generalization of the CWC construction.

Construction C.1 [General CWC.] Let l, L, n, o, t, k be positive integers such that $t \leq L$. (Further restrictions will be placed shortly.) Essentially, l is the length of the input to a PRF (e.g., 128), L is the length of the output from the PRF (e.g., 128), n is the length of the nonce (e.g., 88), o is the length of the offset (e.g., 32), t is the length of the desired tag (e.g., 64 or 128), k is the length of the hash function's keysize (e.g., 127).

Let F be a family of functions from $\{0,1\}^l$ to $\{0,1\}^L$. Let $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ be a family of hash functions with $\mathsf{HashSp}_{\mathcal{HF}} = \{0,1\}^l$ and $\mathsf{KeySp}_{\mathcal{HF}} = \{0,1\}^k$ (and $\mathcal{K}_h$ works by randomly selecting and returning an element from $\{0,1\}^k$ with uniform probability). Let $\mathsf{ctr0} : \mathbb{Z}_{\lceil k/L \rceil} \to \{0,1\}^l$, $\mathsf{ctr1} : \{0,1\}^n \times (\mathbb{Z}_{2^o} - \{0\}) \to \{0,1\}^l$ and $\mathsf{ctr2} : \{0,1\}^n \to \{0,1\}^l$ be efficiently-computable injective functions. If $W = \{\, \mathsf{ctr0}(O) \;:\; O \in \mathbb{Z}_{\lceil k/L \rceil} \,\}$, $X = \{\, \mathsf{ctr1}(N, O) \;:\; N \in \{0,1\}^n, O \in (\mathbb{Z}_{2^o} - \{0\}) \,\}$, $Y = \{\, \mathsf{ctr2}(N) \;:\; N \in \{0,1\}^n \,\}$, and $Z = \{\, \mathcal{H}_K(M) \;:\; K \in \mathsf{KeySp}_{\mathcal{HF}}, M \in \mathsf{MsgSp}_{\mathcal{HF}} \,\}$, we require that W, X, Y, and Z be pairwise mutually exclusive.

Let $\mathsf{extract} : \{0,1\}^{\lceil k/L \rceil \cdot L} \to \{0,1\}^k$ be a function that takes as input a $\lceil k/L \rceil \cdot L$-bit string and that outputs a k-bit string. We require that $\mathsf{extract}$ always pick the same k bits from the input string and always outputs those bits in the exact same order (e.g., $\mathsf{extract}$ returns the first k bits of its input).

Let $\mathcal{SE}[F,\mathcal{HF}] = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ be an AEAD scheme built from function family F and hash function $\mathcal{HF}$ and using the above functions $\mathsf{extract}$, $\mathsf{ctr0}$, $\mathsf{ctr1}$, $\mathsf{ctr2}$. We assume that $\mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{HF}]} \times \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]} \subseteq \mathsf{MsgSp}_{\mathcal{HF}}$ and that all messages in $\mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ have length at most $L \cdot (2^o - 1)$. Note that the former means that the message space of $\mathcal{HF}$ actually consists of pairs of strings. Let $\mathsf{NonceSp}_{\mathcal{SE}[F,\mathcal{HF}]} = \{0,1\}^n$. Let $\mathcal{SE}[F,\mathcal{HF}]$'s component algorithms be defined as follows:

Algorithm $\mathcal{K}_e$
  $f \xleftarrow{\$} F$
  $\boxed{K_h \leftarrow \mathsf{extract}(f(\mathsf{ctr0}(0)) \| f(\mathsf{ctr0}(1)) \| \cdots \| f(\mathsf{ctr0}(\lceil k/L \rceil - 1)))\ ;\ H \leftarrow \mathcal{H}_{K_h}}$
  Return $\langle f, H \rangle$

Algorithm $\mathcal{E}^{N,A}_{\langle f,H \rangle}(M)$
  $\sigma \leftarrow \mathsf{CTR\text{-}MODE}^N_f(M)$
  $\boxed{\tau \leftarrow \text{first } t \text{ bits of } (f(\mathsf{ctr2}(N)) \oplus f(H(A,\sigma)))}$
  Return $\sigma \| \tau$

Algorithm $\mathcal{D}^{N,A}_{\langle f,H \rangle}(C)$
  If $|C| < t$ then return $\mathsf{INVALID}$
  Parse C as $\sigma \| \tau$ // $|\tau| = t$
  If $A \notin \mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ or $\sigma \notin \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ then return $\mathsf{INVALID}$
  $\boxed{\tau' \leftarrow \text{first } t \text{ bits of } (f(\mathsf{ctr2}(N)) \oplus f(H(A,\sigma)))}$
  If $\tau \neq \tau'$ return $\mathsf{INVALID}$
  $M \leftarrow \mathsf{CTR\text{-}MODE}^N_f(\sigma)$
  Return M

Algorithm $\mathsf{CTR\text{-}MODE}^N_f(X)$
  $\alpha \leftarrow \lceil |X|/L \rceil$
  For $i = 1$ to α do
    $Z_i \leftarrow f(\mathsf{ctr1}(N,i))$
  $Y \leftarrow (\text{first } |X| \text{ bits of } Z_1 \| Z_2 \| \cdots \| Z_\alpha) \oplus X$
  Return Y ▌

Remark C.2 Recall that one requirement on the message space for any AEAD scheme is that if it contains any string M, then it contains all strings of length $|M|$. This means that the membership test $\sigma \notin \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ and the application of H to (A,σ) makes sense.

Remark C.3 As specified in the definition, $\mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{HF}]} \times \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]} \subseteq \mathsf{MsgSp}_{\mathcal{HF}}$. This means that we $\mathcal{HF}$ is used to hash *pairs* of strings, not just string. This is not a serious restriction since given any hash function that hashes strings, it is trivial to construct a hash function that hashes pairs of strings (by encoding the pair of strings as a single string in some appropriate manner).

Remark C.4 It is also worth commenting on the purpose of $\mathsf{ctr0}$, $\mathsf{ctr1}$, and $\mathsf{ctr2}$. As shown in Construction C.1, these functions are used to derive the inputs to the construction's underlying function f. By requiring that none of the outputs collide (i.e., that the sets W, X, Y, Z in the definition are pairwise mutually exclusive), we ensure that the inputs to f for different purposes never collide. For example, the inputs to f used for counter mode encryption will always be different than the inputs to f when enciphering the output of H.

Remark C.5 It is possible to obtain a two-key CWC variant by replacing the first boxed step with

$$H \stackrel{\$}{\leftarrow} \mathcal{HF} .$$

C.3 The security of the general CWC construction

We now state the following results for all Construction C.1-style AEAD schemes. We shall prove Lemmas C.6 and C.7 in Appendices C.5 and C.6, respectively.

Lemma C.6 [Integrity of Construction C.1.] Let $\mathcal{SE}[F, \mathcal{HF}]$ be as in Construction C.1 and let $\mathcal{HF}$ be an ϵ-AU hash function. Then given any nonce-respecting AUTH adversary A against $\mathcal{SE}[F, \mathcal{HF}]$, we can construct a PRF adversary B_A against F such that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(B_A) + \epsilon + 2^{-t} .$$

Furthermore, the experiment for B_A takes the same time as the experiment for A and, if A makes at most $q-1$ oracle queries and a total of at most μ bits of payload data (for both these $q-1$ oracle queries and the forgery attempt), then B_A makes at most $\mu/L + 3q + \lceil k/L \rceil$ oracle queries. ▮

Lemma C.7 [Privacy of Construction C.1.] Let $\mathcal{SE}[F, \mathcal{HF}]$ be as in Construction C.1. Then given a nonce-respecting IND\$-CPA adversary A against $\mathcal{SE}[F, \mathcal{HF}]$ one can construct a PRF adversary B_A against F such that

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(B_A) .$$

Furthermore, the experiment for B_A takes the same time as the experiment for A and, if A makes at most q oracle queries totaling at most μ bits of payload data, then B_A makes at most $\mu/L + 3q + \lceil k/L \rceil$ oracle queries. ▮

We interpret these lemmas as follows. Intuitively, the first lemma states that if F is a secure PRF, if $\mathcal{HF}$ is ϵ-AU where ϵ is not too large, and if t is not too small, then $\mathcal{SE}[F, \mathcal{HF}]$ preserves integrity. We comment that most modern block ciphers (e.g., AES) are considered to be secure PRPs (and therefore also secure PRFs up to a birthday term). We also comment that we can construct hash functions $\mathcal{HF}$ with provably small ϵ.

Intuitively, the second lemma states that if F is a secure PRF, then $\mathcal{SE}[F, \mathcal{HF}]$ will preserve privacy. We discuss the meaning of these types of proofs in more detail in Section 4.

C.4 Proof of Theorem 4.1 and Theorem 4.2

The security of the CWC construction from Section 3 follows from Lemmas C.6 and C.7 assuming that (1) CWC as described in Section 3 is really an instantiation of Construction C.1 and (2) that the hash function used in Section 3 is ϵ-AU for some small ϵ. We begin by justifying the second bullet.

Lemma C.8 [**CWC-HASH (Section 3) is ϵ-almost universal.**] Consider the CWC-BC-tl construction from Section 3. Let $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ be the hash function function whose key generation algorithm selects a random key K from $\{0,1\}^{127}$ and let $\mathcal{H}_K$ be the CWC-HASH function except that we replace

$$Z \leftarrow \text{last 127 bits of } \mathsf{BC}_K(110^{126})$$

with

$$Z \leftarrow K \ .$$

Note that $\mathsf{AdSp}_{\mathsf{CWC\text{-}BC\text{-}tl}} \times \mathsf{MsgSp}_{\mathsf{CWC\text{-}BC\text{-}tl}} \subseteq \mathsf{MsgSp}_{\mathcal{HF}}$; that is, $\mathcal{H}_K$ takes two strings as input. Assume $\mathcal{HF}$ hashes pairs of strings where the first string is always at most $n \le \mathsf{MaxAdLen}$ bits long and the second string is always at most $m \le \mathsf{MaxMsgLen}$ bits long. Then $\mathcal{HF}$ is ϵ-almost universal where

$$\epsilon \le \frac{n+m}{2^{133}} + \frac{1}{2^{125}} \ . \ \blacksquare$$

Proof of Lemma C.8: Let (A, σ) and (A', σ') be two inputs to $\mathcal{H}_K$ and let $X = (B_1, \ldots, B_{\beta+1})$ and $Y = (C_1, \ldots, C_{\gamma+1})$ respectively denote their encodings as vectors of 96-bit integers (with $B_{\beta+1}$ and $C_{\gamma+1}$ possibly longer than 96-bits long). Without loss of generality, assume $\beta \le \gamma$ and let $X' = (B'_1, \ldots, B'_{\gamma+1})$ where $B'_i = 0$ for $i \in \{1, \ldots, \gamma - \beta\}$ and $B'_i = B_{i-\gamma+\beta}$ otherwise (i.e., prepend $\gamma - \beta$ zero elements to the X vector).

If $(A, \sigma) \ne (A', \sigma')$ then $X' \ne Y$. This follows from the fact that $B'_{\gamma+1}$ and $C_{\gamma+1}$ respectively encode the lengths of A and σ and of A' and σ' and that if $X' = Y$, then the $B'_{\gamma+1} = C_{\gamma+1}$ and $(A, \sigma) = (A', \sigma')$.

Note that $\mathcal{H}_K(A, \sigma) = \mathcal{H}_K(A', \sigma')$ when

$$\left(B'_1 \cdot K_h^{\gamma} + \cdots + B'_{\gamma} \cdot K_h + B'_{\gamma+1}\right) - \left(C_1 \cdot K_h^{\gamma} + \cdots + C_{\gamma} \cdot K_h + C_{\gamma+1}\right) = 0 \bmod 2^{127} - 1 \qquad (3)$$

where K_h is the hash key derived from K as specified in CWC-HASH. Since the vectors X' and Y are not equal, $\left(B'_1 \cdot K_h^{\gamma} + \cdots + B'_{\gamma} \cdot K_h + B'_{\gamma+1}\right) - \left(C_1 \cdot K_h^{\gamma} + \cdots + C_{\gamma} \cdot K_h + C_{\gamma+1}\right)$ is a non-zero polynomial of degree at most γ. Therefore, by the Fundamental Theorem of Algebra, Equation 3 has at most γ solution modulo $2^{127} - 1$.

Since we are interested in the probability, over the 127-bit keys K, that Equation 3 is true, we note that all keys K_h modulo $2^{127} - 1$ (except 0) have exactly one ways of occurring and that the 0 key can occur in one additional way (i.e., the all 0 string and the all 1 string). This means that of the 2^{127} possible keys K, at most $\gamma + 1$ can lead to keys K_h such that Equation 3 is true.

Finally, note that γ is at most $2 + (n+m)/96$ (the $+2$ comes from the fact that we append 0 bits to A and σ). Consequently

$$\epsilon \le \frac{\frac{n+m}{96} + 3}{2^{127}} \le \frac{n+m}{2^{133}} + \frac{1}{2^{125}}$$

as desired. $\blacksquare$

We now prove Theorem 4.1 and Theorem 4.2, which are corollaries of Lemmas C.6, C.7, and C.8.

Proof of Theorem 4.1 and Theorem 4.2: To prove these theorems we must show that the CWC-BC-tl constructions from Section 3 are instantiations of Construction C.1. We begin by noting that the block cipher BC in CWC-BC-tl plays the role of F in Construction C.1 and that the hash function CWC-HASH (with the simplified key generation algorithm from Lemma C.8) plays the role of $\mathcal{HF}$ in Construction C.1.

Since BC plays the role of F, we have that $l = L = 128$. Furthermore, as described in Section 3, $n = 88$, $o = 32$, $t = \mathsf{tl}$, and $k = 127$. We note that the output the hash function is a 128-bit string whose first bit is always 0. This property, as well as the encodings for the nonce/offsets when encrypting the message and the Carter-Wegman MAC and when generating the hash key, ensure that requisite properties for the interactions between the hash function, $\mathsf{ctr0}$, $\mathsf{ctr1}$, and $\mathsf{ctr2}$.

A direct comparison of the Construction C.1 algorithms and the algorithms from Section 3 shows that they are equivalent. CWC-BC-tl is therefore an instantiation of Construction C.1 and the provable security of CWC-BC-tl follows. ∎

C.5 Proof of Lemma C.6

We being by sketching the proof of Lemma C.6. We first show that applying a random function to the output of an ϵ-AU hash function yields an ϵ'-AXU hash function (Proposition C.10). We then recall the result of Krawczyk [8] that XORing the output of an AXU hash function with a one-time pad yields a secure MAC (Proposition C.12). Such a MAC essentially corresponds to the second and third boxed steps in Construction C.1. (We do not need this final block cipher application if the input to the hash includes the nonce and if we accept a birthday term of the form $q^2\epsilon$.)

We then observe that if we consider a construction like Construction C.1 but with the latter two boxed steps replaced with calls to a secure MAC that tags pairs of strings (A, σ) with nonces N, then that construction would be unforgeable (Proposition C.14). In Proposition C.17 we use the above results to show that $\mathcal{SE}[\mathsf{Func}(l, L), \mathcal{HF}]$ preserves integrity (where $\mathcal{SE}[\mathsf{Func}(l, L), \mathcal{HF}]$ is as in Construction C.1). Lemma C.6 follows.

From AU to AXU. Let us begin with the following construction.

Construction C.9 [Building AXU hash functions from AU hash functions.] Let $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ be a hash function and let $\overline{\mathcal{HF}[t]} = (\overline{\mathcal{K}_h}, \overline{\mathcal{H}})$, t a positive integer, be the hash function defined as follows:

$\overline{\mathcal{K}_h}$	$\overline{\mathcal{H}}_{\langle H,e\rangle}(M)$
$\quad H \stackrel{\$}{\leftarrow} \mathcal{HF}$	$\quad$ Return $e(H(M))$
$\quad e \stackrel{\$}{\leftarrow} \mathsf{Func}(\mathsf{HashSp}_{\mathcal{HF}}, \{0,1\}^t)$	
$\quad$ Return $\langle H, e\rangle$	

Note that $\mathsf{MsgSp}_{\overline{\mathcal{HF}[t]}} = \mathsf{MsgSp}_{\mathcal{HF}}$ and $\mathsf{HashSp}_{\overline{\mathcal{HF}[t]}} = \{0,1\}^t$. ∎

Proposition C.10 Let $\mathcal{HF}$, t, and $\overline{\mathcal{HF}[t]}$ be as in Construction C.9. If $\mathcal{HF}$ is ϵ-AU, then $\overline{\mathcal{HF}[t]}$ is $(\epsilon + 2^{-t})$-AXU.

This result follows from a result in [17, 13] which states that the composition of an ϵ'-AXU hash function, with domain B and range C, with an ϵ-AU hash function, with domain A and range B, is an $(\epsilon+\epsilon')$-AXU hash function with domain A and range C, and the fact that the hash function whose key generation algorithm returns a random function from $\mathsf{Func}(\mathsf{HashSp}_{\mathcal{HF}}, \{0,1\}^t)$ is 2^{-t}-AXU.

Carter-Wegman MACs. Consider now the following construction.

Construction C.11 [Building MACs from AXU hash functions.] Let $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ be a hash function with hash space $\{0,1\}^t$, t a positive integer. We can construct a nonced message authentication scheme $\mathcal{MA} = (\mathcal{K}_m, \mathcal{T}, \mathcal{V})$ as follows:

$\mathcal{K}_m$ $\quad H \xleftarrow{\$} \mathcal{HF}$ $\quad g \xleftarrow{\$} \mathsf{Func}(\mathsf{NonceSp}_{\mathcal{MA}}, \{0,1\}^t)$ $\quad$ Return $\langle H, g\rangle$	$\mathcal{T}_{\langle H,g\rangle}(N, M)$ $\quad$ Return $g(N) \oplus H(M)$	$\mathcal{V}_{\langle H,g\rangle}(N, M, \tau)$ $\quad$ If $g(N) \oplus H(M) = \tau$ then $\quad\quad$ return accept $\quad$ Else return reject

Note that $\mathsf{MsgSp}_{\mathcal{MA}} = \mathsf{MsgSp}_{\mathcal{HF}}$, $\mathsf{TagSp}_{\mathcal{MA}} = \{0,1\}^t$, and that $\mathsf{NonceSp}_{\mathcal{MA}}$ is arbitrary. ▌

We now state the following result, due to Krawczyk [8].

Proposition C.12 Let $\mathcal{HF}$ and $\mathcal{MA}$ be as in Construction C.11. If $\mathcal{HF}$ is ϵ-AXU, then for all nonce-respecting UF adversaries F attacking $\mathcal{MA}$, $\mathbf{Adv}^{\mathrm{uf}}_{\mathcal{MA}}(F) \leq \epsilon$. ▌

As noted in [8], this proposition follows from the facts that XORing the output of the hash function with $g(N)$ prevents any loss of information (assuming that the adversary is nonce-respecting), that a forgery attempt with a previous nonce is upper-bounded by ϵ, and that a forgery attempt with a new nonce is upper-bounded by $2^{-t} \leq \epsilon$.

ENCRYPT-THEN-AUTHENTICATE. Consider the following Encrypt-then-Authenticate [1, 9] construction.

Construction C.13 [Encrypt-then-Authenticate.] Let l, L, n, o, t be positive integers. (Further restrictions will be placed shortly.) Essentially, l is the length of the input to a PRF (e.g., 128), L is the length of the output from the PRF (e.g., 128), n is the length of the nonce (e.g., 88), o is the length of the offset (e.g., 32).

Let F be a family of functions from $\{0,1\}^l$ to $\{0,1\}^L$. Let $\mathcal{MA} = (\mathcal{K}_m, \mathcal{T}, \mathcal{V})$ be a message authentication scheme with $\mathsf{NonceSp}_{\mathcal{MA}} = \{0,1\}^n$ and $\mathsf{TagSp}_{\mathcal{MA}} = \{0,1\}^t$. Let $\mathsf{ctr1} : \{0,1\}^n \times (\mathbb{Z}_{2^o} - \{0\}) \to \{0,1\}^l$ be an efficiently-computable injective function.

Let $\mathcal{SE}[F, \mathcal{MA}] = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ be an AEAD scheme built from function family F and message authentication scheme $\mathcal{MA}$ and using the above function $\mathsf{ctr1}$. We assume that $\mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{MA}]} \times \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{MA}]} \subseteq \mathsf{MsgSp}_{\mathcal{MA}}$ and that all messages in $\mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{MA}]}$ have length at most $L\cdot(2^o-1)$. Note that the former means that the message space of $\mathcal{MA}$ actually consists of pairs of strings. Let $\mathsf{NonceSp}_{\mathcal{SE}[F,\mathcal{MA}]} = \mathsf{NonceSp}_{\mathcal{MA}}$. Let $\mathcal{SE}[F, \mathcal{MA}]$'s component algorithms be defined as follows:

Algorithm $\mathcal{K}_e$
$\quad f \xleftarrow{\$} F$
$\quad \boxed{K \xleftarrow{\$} \mathcal{K}_m}$
$\quad$ Return $\langle f, K\rangle$

Algorithm $\mathcal{E}^{N,A}_{\langle f,K\rangle}(M)$
$\quad \sigma \leftarrow \mathsf{CTR\text{-}MODE}^N_f(M)$
$\quad \boxed{\tau \leftarrow \mathcal{T}^N_K(A, \sigma)}$
$\quad$ Return $\sigma \| \tau$

Algorithm $\mathcal{D}^{N,A}_{\langle f,K\rangle}(C)$
$\quad$ If $|C| < t$ then return $\mathsf{INVALID}$
$\quad$ Parse C as $\sigma \| \tau$ $\quad$ // $|\tau| = t$
$\quad$ If $A \notin \mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{MA}]}$ or $\sigma \notin \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{MA}]}$ then return $\mathsf{INVALID}$
$\quad \boxed{\tau' \leftarrow \mathcal{T}^N_K(A, \sigma)}$

If $\tau \neq \tau'$ return $\mathsf{INVALID}$
$M \leftarrow \mathsf{CTR\text{-}MODE}_f^N(\sigma)$
Return M

Algorithm $\mathsf{CTR\text{-}MODE}_f^N(X)$
$\alpha \leftarrow \lceil |X|/L \rceil$
For $i = 1$ to α do
$Z_i \leftarrow f(\mathsf{ctr1}(N, i))$
$Y \leftarrow$ (first $|X|$ bits of $Z_1 \| Z_2 \| \cdots \| Z_\alpha) \oplus X$
Return Y ▮

Proposition C.14 Let $\mathcal{SE}[F, \mathcal{MA}]$ be as in Construction C.13. Then given a nonce-respecting AUTH adversary B against $\mathcal{SE}[F, \mathcal{MA}]$, we can construct a nonce-respecting forgery adversary D_B against $\mathcal{MA}$ such that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{MA}]}(B) \leq \mathbf{Adv}^{\mathrm{uf}}_{\mathcal{MA}}(D_B) .$$

Furthermore the experiment for D_B uses the same time as the experiment for B and if B makes q encryption oracle queries, then D_B makes q tagging oracle queries. ▮

The approach used in [1] when analyzing Encrypt-then-Authenticate constructions can be used to prove Proposition C.14. The only difference is that we consider MACs that also take nonces as input.

COMBINING THESE CONSTRUCTIONS. Let us now combine these constructions.

Construction C.15 [Combined CWC.] Let l, L, n, o, t, k be positive integers such that $t \leq L$. (Further restrictions will be placed shortly.) Essentially, l is the length of the input to a PRF (e.g., 128), L is the length of the output from the PRF (e.g., 128), n is the length of the nonce (e.g., 88), o is the length of the offset (e.g., 32), t is the length of the desired tag (e.g., 64 or 128), k is the length of the hash function's keysize (e.g., 128).

Let F be a family of functions from $\{0,1\}^l$ to $\{0,1\}^L$. Let $\mathcal{HF} = (\mathcal{K}_h, \mathcal{H})$ be a family of hash functions with $\mathsf{HashSp}_{\mathcal{HF}} = \{0,1\}^l$ and $\mathsf{KeySp}_{\mathcal{HF}} = \{0,1\}^k$ (and $\mathcal{K}_h$ works by randomly selecting and returning an element from $\{0,1\}^k$ with uniform probability). Let $\mathsf{ctr1} : \{0,1\}^n \times (\mathbb{Z}_{2^o} - \{0\}) \rightarrow \{0,1\}^l$ be an efficiently-computable injective function. Let $\mathsf{extract} : \{0,1\}^{\lceil k/L \rceil \cdot L} \rightarrow \{0,1\}^k$ be a function that takes as input a $\lceil k/L \rceil \cdot L$-bit string and that outputs a k-bit string. We require that $\mathsf{extract}$ always pick the same k bits from the input string and always outputs those bits in the exact same order (e.g., $\mathsf{extract}$ returns the first k bits of its input).

Let $\mathcal{SE}[F, \mathcal{HF}] = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ be an AEAD scheme built from function family F and hash function $\mathcal{HF}$ and using the above functions $\mathsf{extract}$ and $\mathsf{ctr1}$. We assume that $\mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{HF}]} \times \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]} \subseteq \mathsf{MsgSp}_{\mathcal{HF}}$ and that all messages in $\mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ have length at most $L \cdot (2^o - 1)$. Note that the former means that the message space of $\mathcal{HF}$ actually consists of pairs of strings. Let $\mathsf{NonceSp}_{\mathcal{SE}[F,\mathcal{HF}]} = \{0,1\}^n$. Let $\mathcal{SE}[F, \mathcal{HF}]$'s component algorithms be defined as follows:

Algorithm $\mathcal{K}_e$
$f \xleftarrow{\$} F$
$\boxed{d \xleftarrow{\$} \mathsf{Func}(\mathbb{Z}_{\lceil k/L \rceil}, \{0,1\}^L) \; ; \; e \xleftarrow{\$} \mathsf{Func}(\mathsf{HashSp}_{\mathcal{HF}}, \{0,1\}^t) \; ; \; g \xleftarrow{\$} \mathsf{Func}(\mathsf{NonceSp}_{\mathcal{SE}[F,\mathcal{HF}]}, \{0,1\}^t)}$
$\boxed{K_h \leftarrow \mathsf{extract}(d(0) \| d(1) \| \cdots \| d(\lceil k/L \rceil - 1)) \; ; \; H \leftarrow \mathcal{H}_{K_h}}$
Return $\langle f, H, e, g \rangle$

Algorithm $\mathcal{E}^{N,A}_{\langle f,H,e,g\rangle}(M)$

$\sigma \leftarrow \mathsf{CTR\text{-}MODE}^N_f(M)$

$\boxed{\tau \leftarrow g(N) \oplus e(H(A,\sigma))}$

Return $\sigma \| \tau$

Algorithm $\mathcal{D}^{N,A}_{\langle f,H,e,g\rangle}(C)$

If $|C| < t$ then return $\mathsf{INVALID}$

Parse C as $\sigma\|\tau$ // $|\tau| = t$

If $A \notin \mathsf{AdSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ or $\sigma \notin \mathsf{MsgSp}_{\mathcal{SE}[F,\mathcal{HF}]}$ then return $\mathsf{INVALID}$

$\boxed{\tau' \leftarrow g(N) \oplus e(H(A,\sigma))}$

If $\tau \neq \tau'$ return $\mathsf{INVALID}$

$M \leftarrow \mathsf{CTR\text{-}MODE}^N_f(\sigma)$

Return M

Algorithm $\mathsf{CTR\text{-}MODE}^N_f(X)$

$\alpha \leftarrow \lceil |X|/L \rceil$

For $i = 1$ to α do

$Z_i \leftarrow f(\mathsf{ctr1}(N,i))$

$Y \leftarrow$ (first $|X|$ bits of $Z_1\|Z_2\|\cdots\|Z_\alpha) \oplus X$

Return Y ▮

Proposition C.16 Let $\mathcal{SE}[F,\mathcal{HF}]$ be as in Construction C.15 and let $\mathcal{HF}$ be an ϵ-AU hash function. Then the advantage of any nonce-respecting AUTH adversary A in breaking the authenticity of $\mathcal{SE}[F,\mathcal{HF}]$ is upper bounded by

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) \leq \epsilon + 2^{-t} \ . \ ▮$$

Proof of Proposition C.16: We first note that the steps $d \xleftarrow{\$} \mathsf{Func}(\mathbb{Z}_{\lceil k/L \rceil}, \{0,1\}^L)$; $K_h \leftarrow \mathsf{extract}(d(0)\|d(1)\|\cdots\|d(\lceil k/L\rceil - 1))$; $H \leftarrow \mathcal{H}_{K_h}$ is equivalent to the step $H \xleftarrow{\$} \mathcal{HF}$.

Note that $e(H(A,\sigma))$ can be rewritten as $\overline{\mathcal{H}}_{\langle H,e\rangle}(A,\sigma)$ where $\overline{\mathcal{HF}[t]} = (\overline{\mathcal{K}_h}, \overline{\mathcal{H}})$ is composed from $\mathcal{HF}$ per Construction C.9.

Also note that $g(N) \oplus \overline{\mathcal{H}}_{\langle H,e\rangle}(A,\sigma)$ can be replaced with $\mathcal{T}^N_{\langle \overline{\mathcal{H}}_{\langle H,e\rangle}, g\rangle}(A,\sigma)$ where $\mathcal{MA} = (\mathcal{K}_m, \mathcal{T}, \mathcal{V})$ is composed from $\overline{\mathcal{HF}[t]}$ as per Construction C.11.

By Proposition C.14, given A we can construct an adversary B_A against $\mathcal{MA}$ such that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) \leq \mathbf{Adv}^{\mathrm{uf}}_{\mathcal{MA}}(B_A) \ .$$

By Proposition C.12 we know that

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathcal{MA}}(B_A) \leq \epsilon'$$

where ϵ' is $\epsilon + 2^{-t}$ (the latter by Proposition C.10). ▮

INTEGRITY OF $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$. We now consider the integrity of $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$.

Proposition C.17 Let $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$ be a AEAD scheme as in Construction C.1. Then for any nonce-respecting AUTH adversary A against $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$, we have that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) \leq \epsilon + 2^{-t} \ . \ ▮$$

Proof of Proposition C.17: Let $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$ be as in Construction C.15. Note that $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$ and $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$ are identical except that the former uses only one random function f and $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$ uses four random functions (one to generate the hash key, one to CTR-mode encrypt the message, one to encipher the output of the hash function, and one to CTR-mode encrypt the output of the hash function). Furthermore, recall that, for $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$, there is never a collision in the input to f between the four different uses of f (this was a requirement imposed on $\mathcal{HF}$, $\mathsf{ctr0}$, $\mathsf{ctr1}$, and $\mathsf{ctr2}$). Consequently, the fact that $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$ uses four random functions and $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$ uses one is immaterial. Hence the probability that A forges against $\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]$ is the same as the probability that it forges against $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$. Ie,

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) = \mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]}(A) .$$

By Proposition C.16, we know the latter probability is upper bounded by $\epsilon + 2^{-t}$. ▌

Proof of Lemma C.6. We now prove Lemma C.6.

Proof of Lemma C.6: Adversary B_A runs A and replies to A's oracle queries using its oracle f. If A returns a valid forgery, B_A returns 1, otherwise B_A returns 0. This implies that

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) = \Pr\left[\, f \xleftarrow{\$} F \;:\; B_A^{f(\cdot)} = 1 \,\right]$$

and

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) = \Pr\left[\, f \xleftarrow{\$} \mathsf{Func}(l,L) \;:\; B_A^{f(\cdot)} = 1 \,\right] .$$

Since

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) \le \epsilon + 2^{-t}$$

by Proposition C.17, we have

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) &= \mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) - \mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) + \mathbf{Adv}^{\mathrm{auth}}_{\mathcal{SE}[\mathsf{Func}(l,L),\mathcal{HF}]}(A) \\
&\le \Pr\left[\, f \xleftarrow{\$} F \;:\; B_A^{f(\cdot)} = 1 \,\right] - \Pr\left[\, f \xleftarrow{\$} \mathsf{Func}(l,L) \;:\; B_A^{f(\cdot)} = 1 \,\right] \\
&\quad + \epsilon + 2^{-t} \\
&= \mathbf{Adv}^{\mathrm{prf}}_{F}(B_A) + \epsilon + 2^{-t}
\end{aligned}$$

as desired. ▌

C.6 Proof of Lemma C.7

Proof of Lemma C.7: Let B_A be a PRF adversary against F that uses adversary A and that has oracle access to a function $g:\ \{0,1\}^l \to \{0,1\}^L$. Adversary B_A runs A and replies to A's encryption oracle queries using its own oracle $g(\cdot)$ for the function f in Construction C.1. Adversary B_A returns the same bit that A returns. Then

$$\Pr\left[\, \langle f,H\rangle \xleftarrow{\$} \mathcal{K}_e \;:\; A^{\mathcal{E}_{\langle f,h\rangle}(\cdot,\cdot,\cdot)} = 1 \,\right] = \Pr\left[\, g \xleftarrow{\$} F \;:\; B_A^{g(\cdot)} = 1 \,\right]$$

since when B_A is given a random instance of F it runs A exactly as if A was given the real encryption oracle. Furthermore

$$\Pr\left[\, A^{\$(\cdot,\cdot,\cdot)} = 1 \,\right] = \Pr\left[\, g \xleftarrow{\$} \mathsf{Func}(l,L) \;:\; B_A^{g(\cdot)} = 1 \,\right]$$

since B_A replies to all of A's oracle queries with independently selected random strings. Consequently

$$\mathbf{Adv}^{\mathrm{priv}}_{\mathcal{SE}[F,\mathcal{HF}]}(A) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(B_A)$$

as desired. ▮

D Test vectors

```
Vector #1:  CWC-AES-128
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:   34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE: 2B 9E AE BE  67 3F AE 03  6B 16 EA 31  DC A7 AE 6B
AES(HVAL):  FC DC 06 4C  CD CA FE E3  DE 7A A3 CF  5C 5D B9 7B
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT: 88 B8 DF 06  28 FD 51 CC  57 55 DB A5  09 9F 3F 1D
            60 04 44 97  DE 89 33 A9

Vector #2:  CWC-AES-192
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:   4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE: 40 E6 24 83  4B 27 9A 7B  15 42 C7 FE  29 EB 29 A3
AES(HVAL):  69 CC 0E 3D  96 98 EB 75  1F 06 A5 90  9B C2 4F 5A
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT: F0 DB A9 74  12 30 01 B0  AF 7A FA 0E  6F 8A D2 3A
            75 8A 1C 43  69 B9 43 28

Vector #3:  CWC-AES-256
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:   35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE: 18 99 E1 A6  1E 6E 37 65  C6 3A 41 99  56 8C D1 BF
AES(HVAL):  1C 56 65 0A  22 BC B5 94  AC F3 CA 24  46 03 B8 5E
```

```
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT: 7B CF 73 BE  46 9C 46 0B  8E 5C 5E 4C  A0 99 A3 65
            F6 50 D1 8A  CB E8 CA FE

Vector #4:  CWC-AES-128
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
            65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE: 2E A9 2A A5  28 B1 1C 08  1C C8 2F 24  9B E4 19 8D
AES(HVAL):  EA 54 F8 3D  56 7F 53 05  88 B1 EA 96  36 79 CD AC
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT: 88 B8 DF 06  28 FD 51 CC  41 DD 25 D4  92 2A 92 FB
            36 CF 0D CE  B4 AD 47 7E

Vector #5:  CWC-AES-192
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
            65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE: 60 3F FC 24  71 64 2E D9  57 E1 B1 EA  F2 F8 B0 34
AES(HVAL):  D8 39 86 2A  33 5A 54 68  C8 16 DA 47  69 A2 10 EB
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT: F0 DB A9 74  12 30 01 B0  1E 8F 72 19  CA 48 6D 27
            A2 9A 63 94  9B D9 1C 99

Vector #6:  CWC-AES-256
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:  00 01 02 03  04 05 06 07
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
            65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE: 0A C6 B1 39  57 7F 26 DA  94 16 42 E1  6D 73 EC B5
AES(HVAL):  4B A5 AD 1E  74 A2 C5 BE  AB D0 DA 4D  F4 29 83 0C
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
```

```
AES(MCPT):  92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT: 7B CF 73 BE  46 9C 46 0B  D9 AF 96 58  F6 87 D3 4F
            F1 73 C1 E3  79 C2 F1 AC

Vector #7:  CWC-AES-128
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:  00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE: 79 00 74 72  E1 C8 36 96  ED 7A B1 F9  03 6E 94 8B
AES(HVAL):  2B 0F 24 69  B1 2B BE 39  C9 40 67 BA  F1 25 E2 5B
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT: 88 B8 DF 06  28 FD 51 CC  31 E6 6E 57  0B 0F 77 80
            86 F9 80 75  7E 7F C7 77  3E 80 E2 73  F1 68 89

Vector #8:  CWC-AES-192
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:  00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE: 2C 5E 3A A4  37 1C 27 D6  E8 6B 76 DC  3D 93 BC 87
AES(HVAL):  48 6E 9C E5  C3 16 3E A6  9C D4 D7 E2  7C 9D 92 D2
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT: F0 DB A9 74  12 30 01 B0  E1 42 B7 58  87 C9 00 8E
            D8 68 D6 3A  04 07 E9 F6  58 6E 31 8E  E6 9E A0

Vector #9:  CWC-AES-256
AES KEY:    00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
            F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:  00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: <None>
NONCE:      FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:   35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE: 4A 70 29 CC  58 25 52 CB  75 AD C9 60  FF B3 F7 55
AES(HVAL):  2B 64 0E 02  CE 51 DE 22  B2 0F 2A 8D  C4 23 CD C0
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT: 7B CF 73 BE  46 9C 46 0B  9B C6 2D DE  26 DD 47 B9
            6E 35 44 4C  74 C8 D3 E8  AC 31 23 49  C8 BF 60
```

```
Vector #10:  CWC-AES-128
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:    34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE: 51 AE 9D 7E  86 BD E0 B2  AA 18 2C 91  87 0A 9C A5
AES(HVAL):   DF 48 30 BD  1D DC E0 59  B1 C2 0B 29  01 4F 80 10
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT: 88 B8 DF 06  28 FD 51 CC  31 E6 6E 57  0B 0F 77 74
             C1 ED 54 D9  89 21 A7 0F  BC EC 71 83  9B 0A C2

Vector #11:  CWC-AES-192
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:    4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE: 51 60 E7 81  DC 64 F9 CD  54 BA 02 40  A2 E8 EE 99
AES(HVAL):   A0 30 58 13  22 B6 80 53  64 B0 3E 52  41 D2 2D 0A
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT: F0 DB A9 74  12 30 01 B0  E1 42 B7 58  87 C9 00 66
             86 AC 20 DB  A4 B9 1C 0E  3C 87 81 B3  A9 21 78

Vector #12:  CWC-AES-256
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E
ASSOC DATA: 54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
---------------------------------------------------------------
HASH KEY:    35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE: 3F F5 0C 60  E6 01 7A 3C  A1 BB B3 54  65 02 85 7C
AES(HVAL):   3E EF A2 E4  97 91 82 86  73 0C F6 E9  46 2C CA 15
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT: 7B CF 73 BE  46 9C 46 0B  9B C6 2D DE  26 DD 47 AC
             E5 99 A2 15  B4 94 77 29  AF ED 47 CB  C7 B8 B5

Vector #13:  CWC-AES-128
```

```
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA: <None>
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:    34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE:  58 D5 28 89  4F 1F 6A 52  A6 44 FA 69  65 C0 73 A6
AES(HVAL):   A3 9E F3 6F  67 1F FA F8  71 0C 83 BB  49 A6 6E BC
MAC CTR PT:  80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT:  88 B8 DF 06  28 FD 51 CC  31 E6 6E 57  0B 0F 77 0F
             48 5B 82 64  6E CF B9 F9  A0 B0 75 4F  D5 94 36 5A
             08 17 2E 86  A3 4A 3B 06  CF 72 64 E3  CB 72 E4 6E

Vector #14:  CWC-AES-192
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA: <None>
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:    4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE:  0D 0A D2 78  1E 8F E8 47  00 85 31 28  B1 E3 49 3A
AES(HVAL):   5A 05 AA 45  88 06 A9 C1  DC 5A F6 AF  6F 8F EC F6
MAC CTR PT:  80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT:  F0 DB A9 74  12 30 01 B0  E1 42 B7 58  87 C9 00 A3
             A4 C4 70 6D  40 41 F4 F9  58 E1 3F D0  D7 60 4D 1E
             9C B3 5E 76  71 14 90 8E  B6 D6 4F 7C  9D F4 E0 84

Vector #15:  CWC-AES-256
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA: <None>
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
-----------------------------------------------------------------
HASH KEY:    35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE:  02 F2 DA E9  83 72 0E BC  DC 77 89 3B  67 CB 3D B7
AES(HVAL):   B7 F6 AE DE  A3 95 35 FE  03 93 08 DF  E0 C7 F1 78
MAC CTR PT:  80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT:  7B CF 73 BE  46 9C 46 0B  9B C6 2D DE  26 DD 47 B5
             D2 41 06 CA  5D EB 80 A7  B5 71 0A 38  A4 39 8D BA
             25 FC 95 98  21 B0 23 0F  59 30 13 71  6D 2C 83 D8
```

```
Vector #16:  CWC-AES-128
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA:  54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
--------------------------------------------------------------
HASH KEY:    34 AE 6A 6F  E9 51 78 94  AC CC BB 9E  BA E7 20 8C
HASH VALUE:  05 EE B6 CB  DF A6 E5 B8  4C 65 DD F4  8C C8 25 23
AES(HVAL):   62 E5 23 FE  48 8F BC 14  E3 77 15 6C  4D 0F D0 8B
MAC CTR PT:  80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   AB 89 DD E9  C4 55 C1 FE  BE 7E E7 58  82 D4 8A D2
CIPHERTEXT:  88 B8 DF 06  28 FD 51 CC  31 E6 6E 57  0B 0F 77 0F
             48 5B 82 64  6E CF B9 F9  A0 B0 75 4F  D5 94 36 5A
             C9 6C FE 17  8C DA 7D EA  5D 09 F2 34  CF DB 5A 59

Vector #17:  CWC-AES-192
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA:  54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
--------------------------------------------------------------
HASH KEY:    4F A8 88 AF  06 83 60 0C  AB 35 75 EF  0A E6 01 A5
HASH VALUE:  10 E1 48 E2  D0 68 39 EC  C4 0A 6C A3  D6 8B 47 54
AES(HVAL):   23 0A 37 C3  48 7C 9F 76  05 B9 5D 1A  21 D5 D5 FD
MAC CTR PT:  80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):   C6 B6 F4 33  F9 12 39 4F  6A 8C B9 D3  F2 7B 0C 72
CIPHERTEXT:  F0 DB A9 74  12 30 01 B0  E1 42 B7 58  87 C9 00 A3
             A4 C4 70 6D  40 41 F4 F9  58 E1 3F D0  D7 60 4D 1E
             E5 BC C3 F0  B1 6E A6 39  6F 35 E4 C9  D3 AE D9 8F

Vector #18:  CWC-AES-256
AES KEY:     00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             F0 E0 D0 C0  B0 A0 90 80  70 60 50 40  30 20 10 00
PLAINTEXT:   00 01 02 03  04 05 06 07  08 09 0A 0B  0C 0D 0E 0F
             80 81 82 83  84 85 86 87  88 89 8A 8B  8C 8D 8E 8F
ASSOC DATA:  54 68 69 73  20 69 73 20  61 20 70 6C  61 69 6E 74
             65 78 74 20  68 65 61 64  65 72 2E 00
NONCE:       FF EE DD CC  BB AA 99 88  77 66 55
--------------------------------------------------------------
HASH KEY:    35 8F 2B 0C  FF E9 84 BE  F9 EE EE 55  85 36 BC E5
HASH VALUE:  09 4D C5 21  94 79 E0 58  4E E9 C1 2C  29 6A E3 A4
AES(HVAL):   E9 69 49 47  09 07 62 3B  A9 8D AD 51  9F D5 D1 F7
```

```
MAC CTR PT: 80 FF EE DD  CC BB AA 99  88 77 66 55  00 00 00 00
AES(MCPT):  92 0A 3B 46  82 25 16 F1  5A A3 1B AE  8D EB 72 A0
CIPHERTEXT: 7B CF 73 BE  46 9C 46 0B  9B C6 2D DE  26 DD 47 B5
            D2 41 06 CA  5D EB 80 A7  B5 71 0A 38  A4 39 8D BA
            7B 63 72 01  8B 22 74 CA  F3 2E B6 FF  12 3E A3 57
```