An extended abstract of this paper appears in *Fast Software Encryption*, Lecture Notes in Computer Science Vol. ??, W. Meier and B. Roy eds., Springer-Verlag, 2004. This is the full version.

CWC: A high-performance conventional authenticated encryption mode

Tadayoshi Kohno* John Viega† Doug Whiting‡

January 15, 2004

Abstract

We introduce CWC, a new block cipher mode of operation for protecting both the privacy and the authenticity of encapsulated data. CWC is currently the only such mode having all five of the following properties: provable security, parallelizability, high performance in hardware, high performance in software, and no intellectual property concerns. We believe that having all five of these properties makes CWC a powerful tool for use in many performance-critical cryptographic applications. CWC is also the only appropriate solution for some applications; e.g., standardization bodies like the IETF and NIST prefer patent-free modes, and CWC is the only such mode capable of processing data at 10Gbps in hardware, which will be important for future IPsec (and other) network devices. As part of our design, we also introduce a new parallelizable universal hash function optimized for performance in both hardware and software.

Keywords: Authenticated encryption, modes of operation, parallelism, performance, security proofs.

*Dept. of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-mail: `tkohno@cs.ucsd.edu`. URL: `http://www-cse.ucsd.edu/users/tkohno` Supported by a National Defense Science and Engineering Graduate Fellowship.

†Virginia Tech, 6066 Leesburg Pike, Suite 500, Falls Church, VA 22041, USA. E-mail: `viega@securesoftware.com`. URL: `http://www.viega.org/`.

‡Hifn, Inc., 5973 Avenida Encinas, Suite 110, Carlsbad, CA 92009, USA. E-mail: `dwhiting@hifn.com`.

1 Introduction

An *authenticated encryption associated data* (AEAD) scheme is a symmetric encryption scheme designed to protect both the privacy and the authenticity of encapsulated data. There has recently been a strong push toward producing block cipher-based AEAD schemes [13, 10, 12, 24, 29, 23, 5]. Despite this push, among the previous works there does not exist any AEAD scheme simultaneously having all of the following properties: provable security, parallelizability, high performance in hardware, high performance in software, and free from intellectual property concerns. Even though not all applications will require all five of the these properties, almost all applications will require at least one of the them, and may very likely have to be able to interoperate with an application requiring a different property. We thus view finding an appropriate scheme having all five of these properties as a very important research goal.

Finding an appropriate balance between all five of the aforementioned properties is, however, not easy because the most natural approaches to addressing some of the properties are actually disadvantageous with respect to other properties. We believe we have overcome these challenges and, in doing so, introduce a new mode of operation called CWC, or Carter-Wegman Counter mode.

MOTIVATING EXAMPLE. One of the primary motivations for such a block cipher-based AEAD scheme is IPsec. From a pragmatic perspective, we note that many vendors and standardization bodies prefer patent-free modes over patented modes (the elegant OCB mode was apparently rejected from the IEEE 802.11 working group because of patent concerns). And, from a hardware performance perspective, we note that because none of the existing patent-free AEAD schemes are parallelizable, it to impossible to make existing patent-free AEAD schemes run faster than about 2Gbps using conventional ASIC technology and a single processing unit. Nevertheless, future network devices will be expected to run at 10Gbps. CWC addresses these issues, being both patent-free and capable of processing data at 10Gbps using conventional ASIC technology.

THE CWC SOLUTION. Our new mode of operation, called CWC, has all five of the properties mentioned above. It is provably secure. Moreover, our provable security-based analyses helped guide our research and helped us reject other schemes with similar performance properties but with slightly worse provable security bounds. CWC is also parallelizable, which means that we can make CWC-AES run at 10Gbps using conventional ASIC technology. CWC is also fast in software. Our current implementation of CWC-AES runs at about the same speed as the other patent-free modes on 32-bit architectures (Table 1), and we anticipate significant performance gains on 32-bit CPUs when using more sophisticated implementation techniques (Section 6), and we also see significantly better performance on 64-bit architectures. Of course, we do remark that the patented modes like OCB are capable of running even faster in software, which would make them very attractive were they not also encumbered in intellectual property issues.

Like the other two unpatented block cipher-based AEAD modes, CCM [29] and EAX [5], CWC avoids patents by using two inter-related but mostly independent modules: one module to "encrypt" the data and one module to "authenticate" the data. Adopting the terminology used in [5], it is because of the two-module structure that we call CWC a "conventional" block cipher-based AEAD scheme. Although CWC uses two modules, it can be implemented efficiently in a single pass. By using the conventional approach, CCM, EAX, and CWC are very much like composition-based AEAD scheme [4, 15], or AEAD schemes composed from existing encryption schemes and MACs. Unlike composition-based AEAD schemes, however, by designing CWC directly from a block cipher, we eliminate redundant steps and fine-tune CWC for efficiency, again keeping in mind both our hardware and software goals. For example, we use only one block cipher key, which saves expensive memory access in hardware.

The encryption core of CWC is essentially counter (CTR) mode encryption, which is well-known

	Linux/gcc-3.2.2					Windows 2000/Visual Studio 6.0				
	Payload message lengths (bytes)					Payload message lengths (bytes)				
Mode	128	256	512	2048	8192	128	256	512	2048	8192
CWC-AES	105.5	88.4	78.9	72.2	70.5	84.7	70.2	62.2	56.5	55.0
CCM-AES	97.9	87.1	82.0	78.0	77.1	64.8	56.7	52.5	49.5	48.7
EAX-AES	114.1	94.9	86.1	79.1	77.5	75.2	61.8	55.3	50.4	49.1

Table 1: Software performance (in clocks per byte) for the three patent-free block cipher-based AEAD modes on a Pentium III. All implementations were in C and written by Brian Gladman [9] and use 128-bit AES keys. Values are averaged over 50 000 samples. We do not include software performance for the patented modes, like OCB, in this table; the performance for these modes in software will be approximately twice as fast as the shown measurements. Please see the text for additional information and discussion.

---

to be efficient and parallelizable. Finding an appropriate algorithm for the authentication core of CWC proved to be more of a challenge. For authentication, we decided to base our design on the Carter-Wegman [28] universal hash function approach for message authentication. Part of the difficulty in the design came down to choosing the right type of universal hash function, with the right parameters. Since polynomial evaluation can be parallelized (if the polynomial is in x, one can split it into i polynomials in x^i), we chose to use a universal hash function consisting of evaluating a polynomial modulo the prime $2^{127}-1$. We note the our hash function is similar to Bernstein's hash127 [6] except that Bernstein's hash function was optimized for software performance at the expense of hardware performance. To address this issue, we use larger coefficients than Bernstein uses. We believe our hardware- and software-optimized universal hash function to be of independent interest.

Notation. As part of our research, we first created a general approach for combining CTR mode encryption with a universal hash function in order to provide authenticated encryption. We shall refer to this general approach as CWC (note no change in font), and shall use CWC-BC to refer to a CWC instantiation with a 128-bit block cipher BC as the underlying block cipher and with the universal hash function described briefly above. We shall use CWC as shorthand for CWC-BC and use CWC-AES to mean CWC-BC with AES [8] as the underlying block cipher. Other instantiations of the general CWC approach are possible, e.g., for legacy 64-bit block ciphers. Since we are primarily targeting new applications, and since a mode using a 128-bit block cipher will never be asked to interoperate with a mode using a 64-bit block cipher, we focus this paper only on our 128-bit CWC instantiation.

When we say that an AEAD scheme's encryption algorithm takes a pair (A, M) as input and produces a ciphertext as output, we mean that the AEAD scheme is designed to protect the privacy of M and the authenticity of both A and M. This will be made more formal in the body.

Performance. Let (A, M) be some input to the CWC encryption algorithm. The CWC encryption algorithm derives a universal hash subkey from the block cipher key. Assuming that the universal hash subkey is maintained across invocations, encrypting (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 6). As noted above, we could evaluate it in parallel. Serially, assuming no precomputation, we could evaluate this polynomial using d 127x127-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 $d - m$ 96x127-bit multiplies and m 127x127-bit multiplies, where $m = \lceil (d+1)/n \rceil - 1$.

In hardware using conventional ASIC technology at 0.13 micron, it takes approximately 300 Kgates to reach 10 Gbps throughput for CWC-AES. This is around twice as much as OCB, but avoids IP negotiation overhead and royalty payments to three parties. Table 1 relates the software performance, on a Pentium III, of CWC-AES to the two other patent-free AEAD modes CCM and EAX; the patented modes such as OCB are not included in this table, but are about twice as fast as the times given for the patent-free modes. The implementations used to compute Table 1 were written in C by Brian Gladman [9] and all use 128-bit AES keys; the current CWC-AES implementation does not use the above-mentioned precomputation approach for evaluating the polynomial. Table 1 shows that the current implementations of the three modes have comparable performance in software, the relative "best" depending on the OS/compiler and the length of the message. Using the above-mentioned precomputation approach and switching to assembly, 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 compared to CCM-AES and EAX-AES (since the authentication portions of CCM-AES and EAX-AES are limited by the speed of AES but the authentication portion of CWC-AES is limited by the speed of the universal hash function). For comparison, Bernstein's related hash127, which also evaluates a polynomial modulo $2^{127} - 1$ but whose specific structure makes it less attractive in hardware, runs around 4 cpb on a Pentium III when written in assembly and using the precomputation approach. On 64-bit G5s, our initial implementation of the hash function runs at around 6 cpb, thus showing that CWC-AES is very attractive on 64-bit architectures (when running the G5 in 32-bit mode, our implementation runs at around 15 cpb).

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 AEAD scheme for such low-end processors.

The patent issue. The patent issue is a very peculiar one. While it may initially sound odd to let patents influence research, we note that it is also not uncommon, especially in other sciences. Indeed, we view this line of research as discovering the most appropriate solution given real-world constraints. And, just like performance constraints, intellectual property constraints are very real.

Background and related work. The notion of an *authenticated encryption (AE) scheme* was formalized by Katz and Yung [13] and by Bellare and Namprempre [4] and the notion of an *authenticated encryption with associated data (AEAD) scheme* was formalized by Rogaway [23]. Bellare and Namprempre [4] and Krawczyk [15] 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 [13], XECB [10], IAPM [12], OCB [24], CCM [29], and EAX [5]. CWC is similar to the combination of McGrew's UST [20] and TMMH [19], where one of the main advantages of CWC over UST+TMMH is CWC's small key size, which, as the author of UST and TMMH noted, can be a bottleneck for UST+TMMH in hardware at high speeds. The integrity portion of CWC builds on top of the Carter-Wegman universal hashing approach to message authentication [28]. The specific hash function CWC uses is similar to Bernstein's hash127 [6], but is better suited for hardware. Shoup [26] and Nevelsteen and Preneel [21] also worked on software optimizations for universal hash functions. Rogaway and Wagner released a critique of CCM [25]. For each issue raised in [25], we find that we have 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; we stress, however, that, if desired, it is easy to modify CWC to handle arbitrary bit-length messages, see Section 5).

2 Preliminaries

Notation. If x is a string then $|x|$ denotes its length in bits. 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 strings, then $x \| y$ denotes their concatenation. 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, $\mathsf{toint}(10000010) = 2^7 + 2 = 130$. If b is a bit and n a non-negative integer, then b^n denote b concatenated with itself n times; e.g., 10^7 is the string 10000000. 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 such time references.

Authenticated encryption schemes with associated data. We use Rogaway's notion of an *authenticated encryption with associated data (AEAD) scheme* or *mode* [23]. 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 (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 $K \stackrel{\$}{\leftarrow} \mathcal{K}_e$. 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, $|\mathcal{E}_K^{N,A}(M)| = l(|M|)$.

Under the correct usage of an AEAD scheme, after a random key is selected, the application should never invoke the encryption algorithm twice with the same nonce value until a new key is randomly selected. In order to ensure that a nonce does not repeat, implementations typically use nonces that contain counters. We use the notion of a nonce, rather than simply a counter, because the notion of a nonce is more general and allows the developer the freedom to structure the nonce as he or she desires.

Block ciphers. A block cipher $E: \{0,1\}^k \times \{0,1\}^L \to \{0,1\}^L$ is a function from k-bit keys and L-bit blocks to L-bit blocks. We use $E_K(\cdot)$, $K \in \{0,1\}^k$, to denote the function $E(K, \cdot)$ and we use $f \stackrel{\$}{\leftarrow} E$ as short hand for $K \stackrel{\$}{\leftarrow} \{0,1\}^k$; $f \leftarrow E_K$. Block ciphers are families of permutations; namely, for each key $K \in \{0,1\}^k$, E_K is a permutation on $\{0,1\}^L$. We call k the key length of E and we call L the block length.

We adopt the notion of security for block ciphers introduced in [17] and adopted for the concrete setting in [2]. Let $E: \{0,1\}^k \times \{0,1\}^L \to \{0,1\}^L$ be a block cipher and let $\mathsf{Perm}(L)$ denote the set of all permutations on $\{0,1\}^L$. Let A be an adversary with access to an oracle and that returns a bit. Then

$$\mathbf{Adv}_F^{\mathrm{prp}}(A) = \Pr\left[f \xleftarrow{\$} E \ :\ A^{f(\cdot)} = 1 \right] - \Pr\left[g \xleftarrow{\$} \mathsf{Perm}(L) \ :\ A^{g(\cdot)} = 1 \right]$$

denotes the PRP-advantage of A in distinguishing a random instance of E from a random permutation. Intuitively, we say that E is a secure PRP, or a secure block cipher, if the PRP-advantages of all adversaries using reasonable resources is small. Modern block ciphers, such as AES [8], are believed to be secure PRPs.

3 The CWC mode of operation

We now describe our new AEAD scheme. Let $\mathsf{BC} : \{0,1\}^{\mathsf{kl}} \times \{0,1\}^{128} \to \{0,1\}^{128}$ be a 128-bit block cipher. Let $\mathsf{tl} \leq 128$ is the desired tag length in bits. Then the CWC mode of operation using BC with tag length tl, $\mathsf{CWC\text{-}BC\text{-}tl} = (\mathcal{K}, \mathsf{CWC\text{-}ENC}, \mathsf{CWC\text{-}DEC})$, is defined as follows. The message spaces are:

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

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

The $\mathsf{CWC\text{-}BC\text{-}tl}$ key generation, encryption, and decryption algorithms are defined as follows:

Algorithm $\mathcal{K}$
- $K \xleftarrow{\$} \{0,1\}^{\mathsf{kl}}$
- Return K

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

Algorithm $\mathsf{CWC\text{-}DEC}_K(N, A, C)$
- If $|C| < \mathsf{tl}$ then return $\mathsf{INVALID}$
- 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 $\mathsf{INVALID}$
- If $\tau \neq \mathsf{CWC\text{-}MAC}_K(N, A, \sigma)$ then return $\mathsf{INVALID}$
- $M \leftarrow \mathsf{CWC\text{-}CTR}_K(N, \sigma)$
- Return M

The remaining algorithms ($\mathsf{CWC\text{-}CTR}$, $\mathsf{CWC\text{-}MAC}$, $\mathsf{CWC\text{-}HASH}$) are defined below. The $\mathsf{CWC\text{-}CTR}$ algorithm handles generating the encryption and decryption keystreams, $\mathsf{CWC\text{-}MAC}$ handles the generation of an authentication tag, and uses $\mathsf{CWC\text{-}HASH}$ as the underlying universal hash function.

Algorithm $\mathsf{CWC\text{-}CTR}_K(N, M)$
- $\alpha \leftarrow \lceil |M|/128 \rceil$
- For $i = 1$ to α do
  - $s_i \leftarrow \mathsf{BC}_K(10^7 \| N \| \mathsf{tostr}(i, 32))$
- $x \leftarrow$ first $|M|$ bits of $s_1 \| s_2 \| \cdots \| s_\alpha$
- $\sigma \leftarrow x \oplus M$
- Return σ

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

Algorithm $\mathsf{CWC\text{-}HASH}_K(A, \sigma)$
- $Z \leftarrow$ last 127 bits of $\mathsf{BC}_K(110^{126})$
- $K_h \leftarrow \mathsf{toint}(Z)$
- $l \leftarrow$ min integer such that 96 divides $|A \| 0^l|$
- $l' \leftarrow$ min integer such that 96 divides $|\sigma \| 0^{l'}|$
- $X \leftarrow A \| 0^l \| \sigma \| 0^{l'}$; $\beta \leftarrow |X|/96$
- Break X into chunks $X_1, X_2, \ldots, X_\beta$
- For $i = 1$ to β do
  - $Y_i \leftarrow \mathsf{toint}(X_i)$
- $l_\sigma \leftarrow |\sigma|/8$; $l_A \leftarrow |A|/8$
- $Y_{\beta+1} \leftarrow 2^{64} \cdot l_A + l_\sigma$
- $R \leftarrow Y_1 K_h^\beta + \cdots + Y_\beta K_h + Y_{\beta+1} \bmod 2^{127} - 1$
- Return $\mathsf{tostr}(R, 128)$

4 Theorem statements

The CWC scheme is a provably secure AEAD scheme assuming that the underlying block cipher, e.g., AES, is a secure 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 at least the same assumptions we make, and some modes, such as OCB [24], require the stronger, albeit still reasonable, assumption of super-pseudorandomness.

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

4.1 Privacy

We first show that if BC is a secure block cipher, then CWC-BC-tl will preserve privacy under chosen-plaintext attacks. For our notion of privacy for AEAD schemes, we use the strong definition of indistinguishability from [23]. Let $\mathcal{SE} = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ be an AEAD scheme with length function $l(\cdot)$. 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

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

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.

Theorem 4.1 [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 PRP adversary C_A against BC such that if A makes at most q oracle queries totaling at most μ bits of payload message data, then

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

Furthermore, the experiment for C_A takes the same time as the experiment for A and C_A makes at most $\mu/128 + 3q + 1$ oracle queries. ▌

Let us elaborate on why Theorem 4.1 implies that CWC-BC will preserve privacy under chosen-plaintext attacks. 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 resources and, in particular, this means that, for C_A as described in the 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 are small, then, because of the above equations, $\mathbf{Adv}^{\mathrm{priv}}_{\mathsf{CWC\text{-}BC\text{-}tl}}(A)$ must also be small as well. I.e., any adversary A using reasonable resources will only be able to break the privacy of CWC-BC-tl with some small probability.

As a concrete example, let us consider limiting the number of applications of CWC-BC-tl between rekeyings to some reasonable value such as $q = 2^{32}$, and let us limit the total number of payload bits between rekeyings to $\mu = 2^{50}$. Then Equation 1 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}.

4.2 Integrity/authenticity

We now present our results showing that if BC is a secure block cipher, then CWC-BC-tl will protect the authenticity of encapsulated data. We use the strong notion of authenticity for AEAD schemes from [23]. Let $\mathcal{SE} = (\mathcal{K}_e, \mathcal{E}, \mathcal{D})$ be an AEAD scheme. 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, the scheme $\mathcal{SE}$ preserves integrity/authenticity if the AUTH-advantage of all nonce-respecting adversaries using reasonable resources is small.

Theorem 4.2 [Integrity/authenticity of CWC.] Let CWC-BC-tl be as specified in Section 3. (Recall that BC is a 128-bit block cipher and that the tag length tl is ≤ 128.) Consider a nonce-respecting AUTH adversary A against CWC-BC-tl. Assume the execution environment allows A to query its oracle with associated data that are at most $n \leq \mathsf{MaxAdLen}$ bits long and with messages that are at most $m \leq \mathsf{MaxMsgLen}$ 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 PRP adversary C_A against BC such that

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

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

Let us elaborate on why Theorem 4.2 implies that CWC-BC will preserve authenticity. 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 resources and, in particular, this means that, for C_A as described in the 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 2 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 set $q = 2^{32}$ and $\mu = 2^{50}$ as before, 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}.

Remark 4.3 [Chosen-ciphertext privacy.] Since CWC-BC-tl preserves privacy under chosen-plaintext attacks (Theorem 4.1) *and* provides integrity (Theorem 4.2) assuming that BC is a secure pseudorandom permutation, it also provides privacy under chosen-ciphertext attacks under the same assumption about BC. See [4, 23] for a discussion of the relationship between chosen-plaintext privacy, integrity, and chosen-ciphertext privacy; this relationship was also used, for example, by the designers of OCB [24].

5 Design decisions

Finding an appropriate balance between provable security, hardware efficiency, and software efficiency, while simultaneously avoiding existing intellectual property issues, proved to be one the the biggest challenges of this research project. In this section we discuss how our diverse set of goals affected our design decisions.

THE CWC-HASH UNIVERSAL HASH FUNCTION. We found that the best way to simultaneously achieve our parallelizability, hardware, and software goals was to base the authentication portion of CWC on the Carter-Wegman [28] universal hash function approach to message authentication. This is because universal hash functions, and especially the one we created for CWC, can be implemented in a multitude of ways, thus allowing different platforms and applications to implement CWC-HASH in the way most appropriate for them. For example, hardware implementations will like parallelize the computation of CWC-HASH by splitting it into multiple polynomials in K_h^i for some i. In more detail, if the polynomial is

$$Y_1K_h^{\beta} + Y_2K_h^{\beta-1} + Y_3K_h^{\beta-2} + Y_4K_h^{\beta-3} + \cdots + Y_{\beta}K_h + Y_{\beta+1} \bmod 2^{127} - 1 \ .$$

then, setting $i = 2$, and $y = K_h^2 \bmod 2^{127} - 1$, and assuming β is odd for illustration purposes, we can rewrite the above polynomial as

$$\Big(Y_1y^m + Y_3y^{m-1} + \cdots + Y_{\beta}\Big)x + \Big(Y_2y^m + Y_4y^{m-1} + \cdots + Y_{\beta+1}\Big) \bmod 2^{127} - 1 \ ,$$

After splitting the polynomial, hardware implementations will then likely compute each polynomial using Horner's rule (e.g., the polynomial $aK_h^{2i} + bK_h^i + c$ would be evaluated as $(((a)K_h^i + b)K_h^i) + c)$. Software implementations on modern CPUs, for which memory is cheap, will likely precompute a number of powers of K_h and evaluate the CWC-HASH polynomial directly, or almost directly, using a hybrid between a precomputation approach and Horner's rule. We consider a number of possible implementation strategies in more detail in Section 6.

CWC-HASH is an instantiation of the classic polynomial universal hash approach to message authentication [28], and is closely related to Bernstein's hash127 [6], which also evaluates a polynomial modulo $2^{127} - 1$. Although hash127 is very fast in software, its structure makes it less suitable for use on high-speed hardware. In particular, Bernstein's choice of 32-bit coefficients, while great for software implementations with precomputed powers of K_h, means that hardware implementations using Horner's rule will be "wasting work." Specifically, even with 32-bit coefficients, incorporating each new coefficient using Horner's rule will require a 127x127-bit multiply because the accumulated value will be 127 bits long. By defining the CWC-HASH coefficients to be 96-bits long, we increase the performance of Horner's rule implementations by a factor of three. (Of course, we could have gone even further and made the coefficients 126 bits long, but doing so would have required considerable additional complexity to perform bit and byte shifting within the coefficients.) An alternative approach for increasing the performance of a serial implementation of Horner's rule would be to reduce the size of the CWC-HASH subkey K_h to 96 bits. We discuss why we rejected this option in more detail later, but remark here that there are already more efficient strategies than Horner's rule for implementing CWC-HASH in software, and that in a parallelized approach the values K_h^i, $i \geq 2$, will most often be full 127-bit values even if K_h is only 96-bits long.

ON USING A SINGLE KEY. From a security perspective, it would have been perfectly acceptable, and in fact more traditional, to make the CWC-CTR block cipher key and the two CWC-MAC block cipher keys independent. Like others [29, 5], however, we acknowledge that there are several important reasons for sharing keys between the encryption and authentication portions of modes such as CWC. One of the most important reasons is simplicity of key management. Indeed, fetching key material can be a major bottleneck in high-speed hardware, and minimizing key material is thus important. This fact is also why we derive the hash subkey from the block cipher key rather than use an independent hash subkey. We could, of course, have defined a mode that derived a number of essentially independent block cipher and hash keys from a single block cipher key, but doing so would either have required more memory or more computation and, because we have proofs that our construction works, would have been unnecessary.

Sharing the block cipher key in the way described above and deriving the hash subkey from the block cipher key did, however, mean that we had to be very careful with our proofs of security. To facilitate our proofs, we took extra care in our design to ensure that there would never be a collision in the plaintext inputs to the block cipher between the different usages of the block cipher. For example, by defining CWC-HASH to produce a 127-bit value as output, we know that the first application of BC to $\mathsf{CWC\text{-}HASH}_K(A,\sigma)$ in CWC-MAC will always have its first bit set to 0. To avoid a collision with the input to the keystream generator, the block cipher inputs in CWC-CTR always have the first two bits set to 10. When using the block cipher to create the hash subkey K_h, the first two bits of the input are set to 11.

On the choice of parameters. Part of this effort involved specifying the appropriate parameters for the CWC encryption mode. Example parameters include the nonce length and the way the nonce is encoded in the input to the block cipher. We chose to fix these parameters for interoperability purposes, but note that our general approach in Appendix C does not have theses parameters fixed. We chose to set the nonce length to 88 bits in order to handle future IPsec sequence numbers. And we chose to set the block counter length to 32 bits in order to allow CWC to be used with IPsec jumbograms and other large packets. We also chose to use big-endian byte ordering for consistency purposes and to maintain compatibility with McGrew's ICM Internet-Draft [18] and the IETF, which strongly favors big-endian byte-ordering.

Handling arbitrary bit-length messages. Since we do not believe that many applications will actually require the ability to encrypt arbitrary bit-length messages, we do not define CWC to take arbitrary bit-length messages as input. That said, we did design CWC in such a way that it will be easy to modify the specification to take arbitrary bit-length messages without affecting interoperability with existing implementations when octet-strings are communicated. For example, one could augment the computation of $Y_{\beta+1}$ in CWC-HASH as follows:

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

Of course, a cleaner approach for handling arbitrary bit-length messages would be to compute $l_A \leftarrow |A|$ and $l_\sigma \leftarrow |\sigma|$ in CWC-HASH. We do not define CWC this way because 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.

64-bit block ciphers. We did not define CWC for use with 64-bit block ciphers because we are targeting future high-speed cryptographic applications. Nevertheless, the general CWC approach in Appendix Ccan be instantiated with 64-bit block ciphers. A 64-bit instantiation may, however, require several uncomfortable tradeoffs; e.g., in the length of the nonce.

On the length of the hash subkey. As noted earlier, it is possible to use smaller subkeys K_h in CWC-HASH (simply truncate $\mathsf{BC}_K(110^{126})$ appropriately). Recall that we have fixed the block length of BC to 128 bits. Let hkl denote the length of the hash subkey in an altered construction.

If $\mathsf{hkl} < 127$, then the upper-bound in Equation 2 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. As noted before, 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.

There may occasionally be reasons to use a CWC variant with hash subkeys even shorter than 96 bits. When these situations arise, caution must be exercised since the use of the shorter hash subkeys could significantly impact security. 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.

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 $\mathsf{BC}_K(10^7\|N\|0^{32}) \oplus \mathsf{BC}_K(\mathsf{CWC\text{-}HASH}_K(A,\sigma))$).

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 2 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 2. 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 may happen in some situations.

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. We could, of course, take another block cipher operation to generate the extra key material, but doing so would defeat the purpose.

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 2, 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.

EAX2. Motivated by EAX2 [5], one possible alternative to CWC might be to use $\mathsf{BC}_K(1110^5 \| N)$ both as the value to encrypt R in CWC-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 CWC-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 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. We do comment, however, that we believe the alternatives discussed in this paragraph are still more attractive than EAX because, like CWC but unlike EAX, these alternatives are parallelizable.

USING EXISTING MACs. We chose not to base the authentication portion of our new mode on XOR-MAC [3] or PMAC [7] because of patent concerns and our software performance requirements and we chose not to base the authentication portion on software-efficient MACs such as HMAC [1] because of our hardware parallelizability requirement.

6 Performance

6.1 Hardware

Since one of our main goals was to achieve high performance in hardware and, in particular, to provide a solution for future 10 Gbps IPsec (and other) network devices, 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. With respect to negotiation costs and royalty payments, we note that despite significant demands, to date the relevant parties have not all offered publicly available IP fee schedules. Given this fact, and given today's silicon costs, we believe that the extra silicon for CWC-AES is probably cheaper overall than the negotiation costs and IP fees required for OCB.

6.2 Software

CWC-AES can also be implemented efficiently in software. Table 1 shows timing information for CWC-AES, as well as CCM-AES and EAX-AES, on a 1.133GHz mobile Pentium III dual-booting RedHat Linux 9 (kernel 2.4.20-8) and Windows 2000 SP2. The numbers in the table are the clocks per byte for different message lengths averaged over 50 000 runs and include the entire time for setting up (e.g., expanding the AES key-schedule) and encrypting. All implementations were in C and written by Brian Gladman [9] and use 128-bit AES keys. The Linux compiler was gcc version 3.2.2; the Windows compiler was Visual Studio 6.0. To be fair, we note that OCB does run at about twice the speeds given in Table 1.

From Table 1 we conclude that the three patent-free modes, as currently implemented by Gladman, share similar software performances. The "best" performing one appears to depend on OS/compiler and the length of the message being processed. On Linux, it appears that CWC-AES performs slightly better than EAX-AES for all message lengths that we tested, and better than CCM-AES for the longer messages, whereas Gladman's CCM-AES and EAX-AES implementations slightly outperform his CWC-AES implementation on Windows for all the message lengths that we tested.

Note, however, that all the implementations used to compute Table 1 were written in C. Furthermore, the current CWC-AES code does not make use of all of the optimization techniques (and in particular precomputation) that we describe below. By switching to assembly and using the additional optimization techniques, we anticipate the speed for CWC-HASH to drop to better than 8 clocks per byte, whereas the speed for the CBC-MAC portion of CCM-AES and EAX-AES will be limited by the speed of AES (the best reported speed for AES on a Pentium III is 14.1 cpb, due to a proprietary library by Helger Lipmaa; Gladman's free hand-optimized Windows assembly implementation runs at 17.5 cpb [16]). Returning to the speed of CWC-HASH, for reference we note that Bernstein's related hash127 [6] runs around 4 cpb on a Pentium III when written in assembly and using the precomputation approach. Bernstein's hash127 also works by evaluating a polynomial modulo $2^{127} - 1$; the main difference is that the coefficients for hash127 are 32 bits long, whereas the coefficients for CWC-HASH are 96 bits long (recall Section 5, which discusses why we use 96-bit coefficients). We also note that the performance of CWC-HASH will increase dramatically on 64-bit architectures with larger multiplies; an initial implementation on a G5 using 64-bit integer operations runs at around 6 cpb (when running the G5 in 32-bit mode, the hash function runs at around 15 cpb).

6.2.1 Implementing CWC-HASH in software

Since the implementation of CWC-HASH is more complicated than the implementation of the CWC-CTR portion of CWC, we devote the rest of this section to discussing CWC-HASH.

Precomputation. As noted in Section 5, there are two general approaches to implementing CWC-HASH in software. The first is to use Horner's rule. The second is to evaluate the polynomial directly, which can be faster if one precomputes powers of the hash key K_h at setup time (here the powers of K_h can be viewed as an expanded key-schedule). In particular, as noted in Section 5, evaluating the polynomial using Horner's rule requires a 127x127-bit multiply for each coefficient, whereas evaluating the polynomial directly using precomputed powers of K_h requires a 96x127-bit multiply for each coefficient. (We discuss elsewhere why we did not make the hash subkey 96-bits, which could have sped up a serial Horner's rule implementation.) The advantage with precomputation was first observed by Bernstein in the context of hash127 [6].

The above description of the precomputation approach assumed that if the polynomial is $Y_1 K_h^{\gamma-1} + \cdots + Y_{\gamma-1} K_h + Y_\gamma$ (i.e., the polynomial has γ coefficients), then we had precomputed the powers of K_h^i for all $i \in \{1, \ldots, \gamma-1\}$. The precomputation approach extends naturally to the case where we have precomputed the powers K_h^j, $j \in \{1, \ldots, n\}$, for some $n \leq \gamma - 1$. For simplicity, first assume that we know the polynomial has a multiple of n coefficients. For such a polynomial, one processes the first n coefficients (to get $Y_1 K_h^{n-1} + \ldots + Y_{n-1} K_h + Y_n$), then multiplies the intermediate result by K_h^n (to get $Y_1 K_h^{2n-1} + \ldots + Y_{n-1} K_h^{n+1} + Y_n K_h^n$). After that, one can continue processing data with the same precomputed values (to get $Y_1 K_h^{2n-1} + \ldots + Y_{2n-1} K_h + Y_{2n}$), and so on. Note that each chunk of n coefficients takes $(n-1)$ 96x127-bit multiplies, and all but the last chunk takes an additional 127x127-bit multiply. Now assume that the number of coefficients m in the polynomial is not necessarily a multiple of n. If m is known in advance, one could first process $m \bmod n$ coefficients, multiply by K_h^n, then process in n-coefficient chunks as before. Alternately, as long as the end of the message is known n coefficients in advance, one could process n-coefficients chunks, and then finish off the final $m \bmod n$ coefficients using Horner's rule. Or, if the number of coefficients in the polynomial is not known until the final coefficient is reached, one could process the message in n-coefficient chunks and then multiply by a precomputed power of K_h^{-1} once the end of the message hash been reached.

Naturally, precomputation requires extra memory, but that is usually cheap and plentiful in a software-based environment. Using 32-bit multiplies, the precomputation approach requires 12 32-bit multiplies per 96-bit coefficient, 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 multiplies and three additions with carry per coefficient, 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, which explains why implementations of CWC-HASH for 64-bit architectures are much faster.

Exploiting the parallelism of some instruction sets. On most 32-bit platforms, it turns out that 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 universal hash functions 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 [6].

As noted before, there are only two main differences between the structure of the polynomials of Bernstein's hash127 and CWC-HASH. The first is that Bernstein uses signed coefficients, whereas CWC-HASH uses unsigned coefficients; this should not 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.

Gladman's C implementation of CWC-HASH uses floating point arithmetic, but uses Horner's rule instead of performing precomputation to achieve extra speed. Nothing about the CWC hash indicates that it should run any worse than half the speed of hash127, if implemented in a similar manner, in assembly, and using the floating point registers and precomputation. This upper-bound paints an encouraging picture for CWC performance, because hash127 on a Pentium III runs around 4 cpb when implemented in assembly and using 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 on the same machine.

Finally, it may be possible to further improve the performance of CWC-HASH. For example, literature from the gaming community [11] indicates that one can use both integer and floating point registers in parallel. Although we have not tested this approach, it seems reasonable to conclude that one might be able to interleave integer operations, and thereby obtain additional speedups.

7 Conclusions

In this work we present CWC, the first AEAD mode that is simultaneously provably secure, parallelizable, efficient in hardware and software, and free from intellectual property concerns. Because of its inherent parallelism, CWC-AES is capable of processing data at 10 Gbps in hardware, making it ideal for use with coming 10 Gbps IPsec network devices. CWC-AES is also efficient in software, with the current implementation on 32-bit CPUs comparable to current implementations of the other patent-free (albeit not parallelizable) modes of operations CCM-AES and EAX-AES. In software, we anticipate significant speedups after switching to assembly and using the precomputation approach for CWC-HASH discussed in Section 6, and we have observed significant performance gains on 64-bit CPUs.

Acknowledgments

We thank Peter Gutmann, David McGrew, Fabian Monrose, Avi Rubin, Adam Stubblefield, and David Wagner for their comments. Additionally, we thank Brian Gladman for helping to validate our test vectors and for working with us to obtain timing information.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *CRYPTO '96*, volume 1109 of *LNCS*, pages 1–15. Springer-Verlag, Aug. 1996.

[2] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proc. of the 38th FOCS*, pages 394–403. IEEE Computer Society Press, 1997.

[3] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In D. Coppersmith, editor, *CRYPTO '95*, volume 963 of *LNCS*, pages 15–28. Springer-Verlag, Aug. 1995.

[4] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer-Verlag, Dec. 2000.

[5] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In W. Meier and B. Roy, editors, *FSE 2004*, LNCS. Springer-Verlag, 2004.

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

[7] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In L. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*. Springer-Verlag, 2002.

[8] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.

[9] B. Gladman. AES and combined encryption/authentication modes, 2003. Available at `http://fp.gladman.plus.com/AES/index.htm`.

[10] V. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In M. Matsui, editor, *FSE 2001*, LNCS. Springer-Verlag, 2001.

[11] 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`.

[12] C. Jutla. Encryption modes with almost free message integrity. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 529–544. Springer-Verlag, May 2001.

[13] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In B. Schneier, editor, *FSE 2000*, volume 1978 of *LNCS*, pages 284–299. Springer-Verlag, Apr. 2000.

[14] H. Krawczyk. LFSR-based hashing and authentication. In Y. Desmedt, editor, *CRYPTO '94*, LNCS. Springer-Verlag, Aug. 1994.

[15] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer-Verlag, Aug. 2001.

[16] H. Lipmaa. AES/Rijndael: speed, 2003. Available at `http://www.tcs.hut.fi/~helger/aes/rijndael.html`.

[17] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Computation*, 17(2), Apr. 1988.

[18] D. McGrew. Integer counter mode, Oct. 2002. Available at `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-icm-01.txt`.

[19] 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`.

[20] D. McGrew. The universal security transform, Oct. 2002. Available at `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-ust-01.txt`.

[21] W. Nevelsteen and B. Preneel. In J. Stern, editor, *EUROCRYPT '99*, volume 1592 of *LNCS*, pages 24–41. Springer-Verlag, 1999.

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

[23] P. Rogaway. Authenticated encryption with associated data. In *Proc. of the 9th CCS*, Nov. 2002.

[24] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proc. of the 8th CCS*, pages 196–205. ACM Press, 2001.

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

[26] V. Shoup. On fast and provably secure message authentication based on universal hashing. In N. Koblitz, editor, *CRYPTO '96*, volume 1109 of *LNCS*, pages 313–328. Springer-Verlag, Aug. 1996.

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

[28] 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.

[29] 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 Intellectual property statement

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

B 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) mode. 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 mode.

SYNCHRONIZATION. Synchronization is based on the nonce. As with other nonce-based AEAD modes, 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 mode. 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 construction 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 mode 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 key length) 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 mode, 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.

C Proofs of Theorem 4.1 and Theorem 4.2

Before proving Theorem 4.1 and Theorem 4.2, we first state results about the general CWC construction (see Lemma C.2 and Lemma C.3 below). We then show how Theorems 4.1 and 4.2 follow from Lemmas C.2 and C.3. 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 $K \xleftarrow{\$} \mathcal{K}_h$. 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 $H \xleftarrow{\$} \mathcal{HF}$ be shorthand for $K \xleftarrow{\$} \mathcal{K}_h\ ;\ H \leftarrow \mathcal{H}_K$.

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 $c \in \{0,1\}^n$,

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

PSEUDORANDOM FUNCTIONS. If X and Y are sets, then $\mathsf{Func}(X, Y)$ denotes the set of all functions from X to Y. If l and L are positive integers, then $\mathsf{Func}(l, L)$ denotes the set of all functions from $\{0,1\}^l$ to $\{0,1\}^L$.

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.

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 $\tau \leftarrow \mathcal{T}^N_K(M)$ 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 $\tau' = \mathcal{T}^N_K(M)$, 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 $\mathcal{V}^N_K(M, \tau) = \mathsf{accept}$ 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 ▮

Before proceeding we make several observations. 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.

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).

It is also worth commenting on the purpose of ctr0, ctr1, and 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.

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.2 and C.3 in Appendices C.5 and C.6, respectively.

Lemma C.2 [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.3 [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 4.2 and Theorem 4.1

The security of the CWC construction from Section 3 follows from Lemmas C.2 and C.3 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.4 [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 $n \leq \mathsf{MaxAdLen}$ bits long and the second string is always at most $m \leq \mathsf{MaxMsgLen}$ bits long. Then $\mathcal{HF}$ is ϵ-almost universal where

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

Proof of Lemma C.4: Let (A,σ) and (A',σ') be two distinct 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 $\beta \leq \gamma$ and let $X' = (B'_1,\ldots,B'_{\gamma+1})$ where $B'_j = 0$ for $j \in \{1,\ldots,\gamma-\beta\}$ and $B'_j = B_{j-\gamma+\beta}$ for $j \in \{\gamma-\beta+1,\ldots,\gamma+1\}$ (i.e., prepend $\gamma-\beta$ zero elements to the X vector).

If $(A,\sigma) \neq (A',\sigma')$ then $X' \neq Y$. 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

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

as desired. $\blacksquare$

We now prove Theorem 4.2 and Theorem 4.1, which are corollaries of Lemmas C.2, C.3, and C.4.

Proof of Theorem 4.2 and Theorem 4.1: 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.4) 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, ctr0, ctr1, and 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.

Finally, we apply the standard PRF-PRP switching technique in order to model the underlying block cipher as a PRP rather than a PRF in Theorem 4.2 and Theorem 4.1. ▮

C.5 Proof of Lemma C.2

We being by sketching the proof of Lemma C.2. 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.6). We then recall the result of Krawczyk [14] that XORing the output of an AXU hash function with a one-time pad yields a secure MAC (Proposition C.8). 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.10). In Proposition C.13 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.2 follows.

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

Construction C.5 [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:

$\overline{\mathcal{K}_h}$	$\overline{\mathcal{H}}_{\langle H,e \rangle}(M)$
$H \xleftarrow{\$} \mathcal{HF}$	Return $e(H(M))$
$e \xleftarrow{\$} \mathsf{Func}(\mathsf{HashSp}_{\mathcal{HF}}, \{0,1\}^t)$	
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.6 Let $\mathcal{HF}$, t, and $\overline{\mathcal{HF}[t]}$ be as in Construction C.5. If $\mathcal{HF}$ is ϵ-AU, then $\overline{\mathcal{HF}[t]}$ is $(\epsilon + 2^{-t})$-AXU.

This result follows from a result in [27, 22] 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.7 [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:

$\mathcal{K}_m$	$\mathcal{T}_{\langle H,g\rangle}(N,M)$	$\mathcal{V}_{\langle H,g\rangle}(N,M,\tau)$
$H \xleftarrow{\$} \mathcal{HF}$ $g \xleftarrow{\$} \mathsf{Func}(\mathsf{NonceSp}_{\mathcal{MA}}, \{0,1\}^t)$ Return $\langle H, g\rangle$	Return $g(N) \oplus H(M)$	If $g(N) \oplus H(M) = \tau$ then return accept 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 [14].

Proposition C.8 Let $\mathcal{HF}$ and $\mathcal{MA}$ be as in Construction C.7. 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 [14], 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 [4, 15] construction.

Construction C.9 [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 $\mathcal{K}_e$
- $f \xleftarrow{\$} F$
- $\boxed{K \xleftarrow{\$} \mathcal{K}_m}$
- Return $\langle f, K\rangle$

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

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

Algorithm $\mathsf{CTR\text{-}MODE}_f^N(X)$
  $\alpha \leftarrow \lceil |X|/L \rceil$
  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.10 Let $\mathcal{SE}[F, \mathcal{MA}]$ be as in Construction C.9. 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. ▌

To prove Proposition C.10, we use the approach in [4] for analyzing Encrypt-then-Authenticate constructions. 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.11 [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 $\mathsf{ctr1} : \{0,1\}^n \times (\mathbb{Z}_{2^o} - \{0\}) \to \{0,1\}^l$ be an efficiently-computable injective function. 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 $\mathsf{extract}$ 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 $\mathcal{E}^{N,A}_{\langle f,H,e,g \rangle}(M)$
  $\sigma \leftarrow \mathsf{CTR\text{-}MODE}_f^N(M)$
  $\boxed{\tau \leftarrow g(N) \oplus e(H(A, \sigma))}$
  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.12 Let $\mathcal{SE}[F,\mathcal{HF}]$ be as in Construction C.11 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.12: 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.5.

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.7.

By Proposition C.10, 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.8 we know that

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

where ϵ' is $\epsilon + 2^{-t}$ (the latter by Proposition C.6). ▮

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.13 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.13: Let $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$ be as in Construction C.11. 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 $\mathcal{HF}$, ctr0, ctr1, and 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 $\mathcal{SE}'[\mathsf{Func}(l,L),\mathcal{HF}]$. I.e.,

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

By Proposition C.12, we know the latter probability is upper bounded by $\epsilon + 2^{-t}$. ▮

Proof of Lemma C.2. We now prove Lemma C.2.

Proof of Lemma C.2: 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

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

and

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

Since

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

by Proposition C.13, we have

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

as desired. ▮

C.6 Proof of Lemma C.3

Proof of Lemma C.3: 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

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

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

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

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

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

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
```