This is a shortened version—no proofs—of an upcoming paper.

A Conventional Authenticated-Encryption Mode

M. Bellare*   P. Rogaway†   D. Wagner‡

April 13, 2003

Abstract

We propose a block-cipher mode of operation, EAX, for authenticated-encryption with associated-data (AEAD). Given a nonce N, a message M, and a header H, the mode protects the privacy of M and the authenticity of both M and H. Strings $N, M, H \in \{0,1\}^*$ are arbitrary, and the mode uses $2\lceil M/n \rceil + \lceil H/n \rceil + \lceil N/n \rceil$ block-cipher calls when these strings are nonempty and n is the block length of the underlying block cipher. Among EAX's characteristics are that it is on-line (the length of a message isn't needed to begin processing it) and a fixed header can be pre-processed, effectively removing the per-message cost of binding it to the ciphertext. EAX is obtained by instantiating a simple generic-composition method, EAX2, and then collapsing its two keys into one. EAX is provably secure under a standard complexity-theoretic assumption. EAX is an alternative to CCM [19], and is likewise patent-free.

1 Introduction

AE and AEAD. Authenticated encryption (AE) schemes are symmetric-key mechanisms by which a message M is a transformed into a ciphertext $\mathcal{C}$ in such a way that $\mathcal{C}$ protects both privacy *and* authenticity. Though AE schemes go back more than 20 years, only recently did AE get recognized as a distinct and significant cryptographic goal [6, 7, 12]. Two factors seem to have triggered this. First was the realization that people had been doing rather poorly when they tried to glue together a traditional (privacy-only) encryption scheme and a message authentication code (MAC) [5, 6, 14]; second was the emergence of a class of AE schemes [11, 17] that did *not* work by gluing together an encryption scheme and a MAC.

Following the emergence of new AE schemes and the analysis of old ones, it was realized that often times not all the data should be encrypted—in many applications we have a mixture of secret and non-secret data, and it would be nice to have a mode of operation that provides privacy for the secret data and authenticity for both types of data. Thus was born the notion of *authenticated-encryption with associated-data* (AEAD) [16]. The non-secret data is called the *associated data* or the *header*.

This document. In this note we propose a new AEAD scheme, called EAX. The mechanism is a "conventional" AEAD scheme, meaning a method that, using a block cipher, makes two passes, one aimed at achieving privacy and one aimed at achieving authenticity. Within this space of conventional schemes, we want to do as well as possible. Doing well entails issues of efficiency, simplicity, elegance, patent avoidance, ease of correct use, and provable-security guarantees.

History and related work. The AEAD scheme known as CCM was recently proposed by Whiting, Housley, and Ferguson [19]. By specifying a conventional, two-pass AEAD scheme, the CCM authors aimed to avoid the Intellectual Property (IP) associated to the new, privacy-and-authenticity-melded schemes. But CCM embodies limitations that have nothing to do with the IP that it works to avoid. A note closely related to the current one discusses these limitations [18]. The current note was motivated by a desire to fix the issues identified in CCM while staying within its two-pass (patent-avoiding) framework.

* Department of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-mail: mihir@cs.ucsd.edu WWW: www-cse.ucsd.edu/users/mihir/

† Department of Computer Science, University of California at Davis, Davis, California 95616, USA; and Department of Computer Science, Faculty of Science, Chiang Mai University, Chiang Mai 50200, Thailand. E-mail: rogaway@cs.ucdavis.edu WWW: www.cs.ucdavis.edu/~rogaway/

‡ Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California 94720, USA. E-mail: daw@cs.berkeley.edu WWW: http://www.cs.berkeley.edu/~daw/

Algorithm $\mathrm{CBC}_K\,(M)$

10 Let $M_1 \cdots M_m \leftarrow M$ where $|M_i| = n$
11 $C_0 \leftarrow 0^n$
12 **for** $i \leftarrow 1$ **to** m **do**
13 $\quad C_i \leftarrow E_K(M_i \oplus C_{i-1})$
14 **return** C_m

Algorithm $\mathrm{CTR}_K^{\mathbf{N}}\,(M)$

20 $m \leftarrow \lceil |M|/n \rceil$
21 $S \leftarrow E_K(\mathbf{N}) \,\|\, E_K(\mathbf{N}+1) \,\|\, \cdots \,\|\, E_K(\mathbf{N}+m-1)$
22 $C \leftarrow M \oplus S$ [first $|M|$ bits]
23 **return** C

Algorithm $\mathrm{pad}\,(M;\, B, P)$

30 **if** $|M| \in \{n, 2n, 3n, \ldots\}$
31 **then return** $M \oplus\!\!\rightarrow B$,
32 **else return** $(M \,\|\, 10^{n-1-(|M| \bmod n)}) \oplus\!\!\rightarrow P$

Algorithm $\mathrm{OMAC}_K\,(M)$

40 $L \leftarrow E_K(0^n);\quad B \leftarrow 2L;\quad P \leftarrow 4L$
41 **return** $\mathrm{CBC}_K(\mathrm{pad}\,(M;\, B, P))$

Algorithm $\mathrm{OMAC}_K^t\,(M)$

50 **return** $\mathrm{OMAC}_K([t]_n \,\|\, M)$

Figure 1: Basic building blocks. The block cipher $E\colon \mathsf{Key} \times \{0,1\}^n \to \{0,1\}^n$ is fixed and $K \in \mathsf{Key}$. For CBC, $M \in (\{0,1\}^n)^+$. For CTR, $M \in \{0,1\}^*$ and $\mathbf{N} \in \{0,1\}^n$. For pad, $M \in \{0,1\}^*$ and $B, P \in \{0,1\}^n$ and $\oplus\!\!\rightarrow$ xors the shorter string into the end of longer one. For OMAC, $M \in \{0,1\}^*$ and $t \in [0..2^n - 1]$ and the multiplication of a number by a string L is done in $\mathrm{GF}(2^n)$.

2 Preliminaries

All strings in this note are over the binary alphabet $\{0,1\}$. For $\mathcal{L}$ a set of strings and $n \geq 0$ a number, we let $\mathcal{L}^n$ and $\mathcal{L}^*$ have their usual meanings. The concatenation of strings X and Y is denoted $X \,\|\, Y$ or simply XY. The string of length 0, called the *empty string*, is denoted ε. If $X \in \{0,1\}^*$ we let $|X|$ denote its length, in bits. If $X \in \{0,1\}^*$ and $\ell \leq |X|$ then the first ℓ bits of X are denoted X [first ℓ bits]. When $X \in \{0,1\}^n$ is a nonempty string and $t \in \mathbb{N}$ is a number we let $X + t$ be the n-bit string that results from regarding X as a nonnegative number x (binary notation, most-significant-bit first), adding x to t, taking the result modulo 2^n, and converting this number back into an n-bit string. If $t \in [0..2^n - 1]$ we let $[t]_n$ denote the encoding of t into an n-bit binary string (msb first, lsb last). If X and P are strings then we let $X \oplus\!\!\rightarrow P$ (the *xor-at-the-end* operator) denote the string of length $\ell = \max\{|X|, |P|\}$ bits that is obtained by prepending $\big||X| - |P|\big|$ zero-bits to the shorter string and then xoring this with the other string. (In other words, xor the shorter string into the *end* of the longer string.) A *block cipher* is a function $E\colon \mathsf{Key} \times \{0,1\}^n \to \{0,1\}^n$ where Key is a finite, nonempty set and $n \geq 1$ is a number and $E_K(\cdot) = E(K, \cdot)$ is a permutation on $\{0,1\}^n$. The number n is called the *block length.* Throughout this note we fix such a block cipher E.

In Figure 1 we define the algorithms CBC, CTR, pad, OMAC (no superscript), and $\mathrm{OMAC}^\bullet$ (with superscript). The algorithms CBC (the CBC MAC) and CTR (counter-mode encryption) are standard. Algorithm pad is used only to define OMAC. Algorithm OMAC [9] is a pseudorandom function (PRF) that is a one-key variant of the algorithm XCBC [8]. Algorithm $\mathrm{OMAC}^\bullet$ is like OMAC but takes an extra argument, the integer t. This algorithm is a "tweakable" PRF [15], tweaked in the most simple way possible.

We explain the notation used in the definition of OMAC. The value of iL (line 40: i an integer in $\{2,4\}$ and $L \in \{0,1\}^n$) is the n-bit string that is obtained by multiplying L by the n-bit string that represents the number i. The multiplication is done in the finite field $\mathrm{GF}(2^n)$, using a canonical polynomial to represent field points. For $n = 128$ we use the polynomial $\mathsf{x}^{128} + \mathsf{x}^7 + \mathsf{x}^2 + \mathsf{x} + 1$. In that case, $2L = L{\ll}1$ if the first bit of L is 0 and $2L = (L{\ll}1) \oplus 0^{120}10000111$ otherwise, where $L{\ll}1$ means the left shift of L by one position (the first bit vanishing and a zero entering into the last bit). The value of $4L$ is simply $2(2L)$.

We have made a small modification to the OMAC algorithm as it was originally presented, changing one of its two constants. Specifically, the constant 4 at line 40 was the constant $1/2$ (the multiplicative inverse of 2) in the original definition of OMAC [9]. The OMAC authors indicate that they will promulgate this modification [10], which slightly simplifies implementations.

3 EAX Goals

We wanted a block-cipher-based, nonce-using AEAD scheme. It should provide both privacy, in the sense of indistinguishability from random bits, and integrity, in the sense of an adversary's inability to produce a new but valid (nonce, header, ciphertext) triple [16]. Nothing should be assumed about the nonces except that they are non-repeating. Security must be demonstrated using the standard, provable-security approach. The scheme should employ no tool beyond a block cipher $E\colon \mathsf{Key} \times \{0,1\}^n \to \{0,1\}^n$ that it is based on. We should assume nothing about E beyond its security in the sense of a pseudorandom permutation (PRP). We expect that E will often be instantiated by AES, but we should make no restrictions in this direction (such as insisting that $n = 128$). The scheme should be simple and natural (so, in particular, it should avoid complicated length-annotation). It should be a "conventional" AEAD scheme, making a separate privacy pass and authenticity pass, using no known IP.

We wanted our AEAD scheme to be flexible in the functionality it provides. It should support arbitrary-length messages: the message space should be $\{0,1\}^*$. The key space of the AEAD should be the key space Key of the underlying block cipher. We wanted to support nonces as long as the block length[1]; that is, the nonce space should include $\{0,1\}^n$. Any tag length $\tau \in [0..n]$ should be possible, to allow each user to select how much security she wants from the integrity guarantees and how many bits she has to pay for this.[2] The above considerations imply that the only user-tunable parameters should be E and τ.

We took on some fairly aggressive performance goals. First, message expansion should be no more than required: the length of the ciphertext (which, following the conventions of [17], excludes the nonce) should be only τ bits more than the length of the plaintext. Implementations should be able to profitably pre-process static associated data; for example, if we have an unchanging header attached to every packet, authenticating this header should have no significant cost after a single pre-computation. There should be an efficient pseudorandom function (PRF) directly accessible through the defined interface of the AEAD scheme—as efficient as other conventional PRFs. Key-setup should be efficient and all block-cipher calls should use the same underlying key, so that we do not incur the cost of key scheduling more than once. For both encryption and decryption, we want to use only the forward direction of the block cipher, so that hardware implementations do not need to implement the decryption functionality of the block cipher. The scheme should be on-line for both the plaintext M and the associated data H, which means that one can process streaming data on-the-fly, using constant memory, not knowing when the stream will stop.

EAX achieves all of goals described above.

4 EAX Algorithm

Fix a block cipher $E\colon \mathsf{Key} \times \{0,1\}^n \to \{0,1\}^n$ and a tag length $\tau \in [0..n]$. These parameters should be fixed at the beginning of a particular session that will use EAX mode. Typically, the parameters would be agreed to in an authenticated manner between the sender and the receiver, or they would be fixed for all time for some particular application. Given these parameters, EAX provides a nonce-based AEAD scheme $\mathrm{EAX}[E,\tau]$ whose encryption algorithm has signature $\mathsf{Key} \times \mathsf{Nonce} \times \mathsf{Header} \times \mathsf{Plaintext} \to \mathsf{Ciphertext}$ and whose decryption algorithm has signature $\mathsf{Key} \times \mathsf{Nonce} \times \mathsf{Header} \times \mathsf{Ciphertext} \to \mathsf{Plaintext} \cup \{\textsc{Invalid}\}$ where Nonce, Header, $\mathsf{Plaintext}$, and $\mathsf{Ciphertext}$ are all $\{0,1\}^*$. The EAX algorithm is specified in Figure 2 and a picture illustrating EAX encryption is given in Figure 3.

5 Discussion

No encodings. We have avoided any nontrivial encoding of multiple strings into a single one.[3] Some other approaches that we considered required a PRF to be applied to what was logically a tuple, like (N,H,C). Doing this raises encoding issues we did not want to deal with because, ultimately, there is no

[1] Here we will over-achieve, allowing a nonce space of $\{0,1\}^*$.

[2] Note that since our AEAD scheme is bit-oriented and not byte-oriented, τ is the number of bits, not bytes, of the tag.

[3] One could view the prefixing of $[t]_n$ to M in the definition of $\mathrm{OMAC}_K^t(M)$ as an encoding, but $[t]_n$ is a constant, fixed-length string, and the aim here is just to "tweak" the PRF. That is very different from needing to encode an arbitrary-length message M and an arbitrary-length header H into a single string, for example.

Algorithm $\mathrm{EAX.Encrypt}_K^{N\,H}(M)$

$\mathbf{N} \leftarrow \mathrm{OMAC}_K^0(N)$
$\mathbf{H} \leftarrow \mathrm{OMAC}_K^1(H)$
$C \leftarrow \mathrm{CTR}_K^{\mathbf{N}}(M)$
$\mathbf{C} \leftarrow \mathrm{OMAC}_K^2(C)$
$\mathit{Tag} \leftarrow \mathbf{N} \oplus \mathbf{C} \oplus \mathbf{H}$
$T \leftarrow \mathit{Tag}$ [first τ bits]
return $\mathcal{C} \leftarrow C \,\|\, T$

Algorithm $\mathrm{EAX.Decrypt}_K^{N\,H}(\mathcal{C})$

if $|\mathcal{C}| < \tau$ **then return** INVALID
Let $C \,\|\, T \leftarrow \mathcal{C}$ where $|T| = \tau$
$\mathbf{N} \leftarrow \mathrm{OMAC}_K^0(N)$
$\mathbf{H} \leftarrow \mathrm{OMAC}_K^1(H)$
$\mathbf{C} \leftarrow \mathrm{OMAC}_K^2(C)$
$\mathit{Tag}' \leftarrow \mathbf{N} \oplus \mathbf{C} \oplus \mathbf{H}$
$T' \leftarrow \mathit{Tag}'$ [first τ bits]
if $T \neq T'$ **then return** INVALID
$M \leftarrow \mathrm{CTR}_K^{\mathbf{N}}(C)$
return M

Figure 2: Encryption and decryption under EAX mode. The plaintext is M, the ciphertext is $\mathcal{C}$, the key is K, the nonce is N, and the header is H. The mode depends on a block cipher E (that CTR and OMAC implicitly use) and a tag length τ.

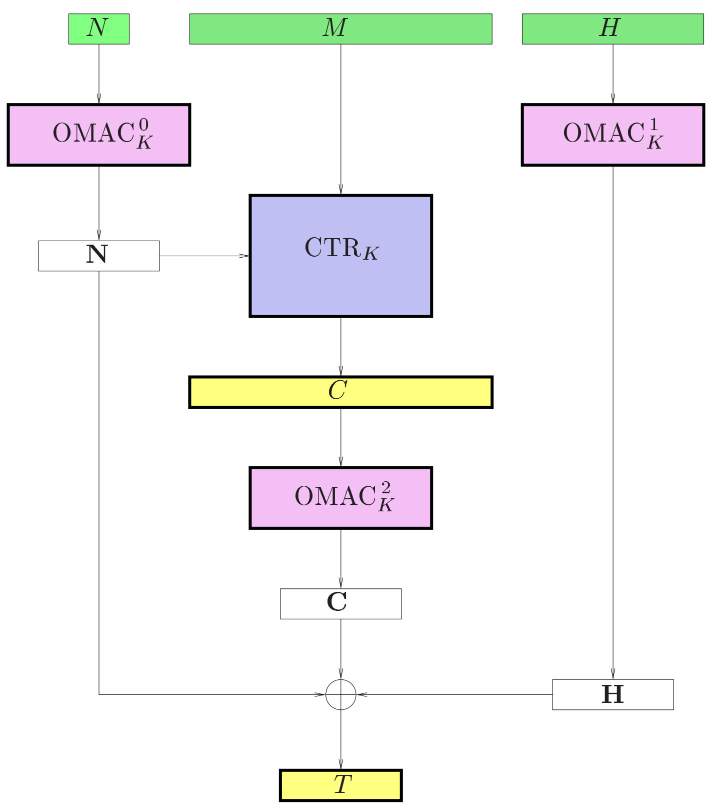

Figure 3: Encryption under EAX mode. The message is M, the key is K, and the header is H. The ciphertext is $C \,\|\, T$.

	CCM	**EAX**
Functionality	Authenticated Encryption with AD	Authenticated Encryption with AD
Built from	Block cipher E with 128-bit blocksize	Block cipher E with n-bit blocksize
Parameters	Block cipher E Tag length $\boldsymbol{\tau} \in \{4, 6, 8, 10, 12, 14, 16\}$ Length of msg length field $\boldsymbol{\lambda} \in [2..8]$	Block cipher E Tag length $\tau \in [0..n]$
Message space	Parameterized: 7 choices: $\boldsymbol{\lambda} \in [2..8]$. Each possible message space a subset of Byte^*, from $\text{Byte}^{2^{16}-1}$ to $\text{Byte}^{<2^{64}-1}$	$\{0,1\}^*$
Nonce space	Parameterized, with a value of $15 - \boldsymbol{\lambda}$ bytes. From 56 bits to 104 bits	$\{0,1\}^*$
Key space	One block-cipher key	One block-cipher key
Ciphertext expansion	$\boldsymbol{\tau}$ bytes	τ bits
Block-cipher calls	$2\left\lceil \frac{\lvert M\rvert}{128} \right\rceil + \left\lceil \frac{\lvert H\rvert}{128} \right\rceil + 2 + \delta$, for $\delta \in \{0,1\}$	$2\left\lceil \frac{\lvert M\rvert}{n} \right\rceil + \left\lceil \frac{\lvert H\rvert}{n} \right\rceil + \left\lceil \frac{\lvert N\rvert}{n} \right\rceil$
Block-cipher calls with static header	$2\left\lceil \frac{\lvert M\rvert}{128} \right\rceil + \left\lceil \frac{\lvert H\rvert}{128} \right\rceil + 2 + \delta$, for $\delta \in \{0,1\}$	$2\left\lceil \frac{\lvert M\rvert}{n} \right\rceil + \left\lceil \frac{\lvert N\rvert}{n} \right\rceil$
Key setup	Block cipher subkeys	Block cipher subkeys 3 block-cipher calls
IV requirements	Non-repeating nonce	Non-repeating nonce
Parallelizable?	No	No
On-line?	No	Yes
Preprocessing (/msg)	Limited (key stream only)	Limited (key stream and header only)
Memory rqmts	Small constant	Small constant
Provable security?	Yes: reduction from block-cipher's PRP security, bound of $\Theta(\sigma^2/2^{128})$	Yes: reduction from block-cipher's PRP security, bound of $\Theta(\sigma^2/2^n)$
Patent-encumbered?	No	No

Figure 4: A comparison of basic characteristics of CCM and EAX.

efficient, compelling, on-line way to encode multiple strings into a single one. Alternatively, one could avoid encodings and consider a new kind of primitive, a multi-argument PRF. But this would be a non-standard tool and we didn't want to use any non-standard tools. All in all, it seemed best to find a way to sidestep the need to do encodings, which is what we have done.

Why not generic composition? Why have we specified a block-cipher based (BC-based) AEAD scheme instead of following the generic-composition approach of combining a (privacy-only) encryption method and a message authentication code? There are reasonable arguments in favor of generic composition, based on aesthetic or architectural sensibilities. One can argue that generic composition better separates conceptually independent elements (privacy and authenticity) and, correspondingly, allows greater implementation flexibility [6, 14]. Correctness becomes much simpler and clearer as well. The argument does have validity. Still, BC-based AEAD modes have some important advantages. BC-based AEAD enables improved efficiency (the strand of work not represented here but found in [11, 17]) and makes it easier to use a cryptosystem correctly and interoperably—for example, presenting a more directly useful API for developers. BC-based AEAD reduces the risk that implementors will choose insecure parameters. It makes it easier for implementors to use a scheme without knowing a lot of cryptography. It saves on key bits and key-setup time, as generic-composition methods invariably require a pair of separate keys. Finally, it was a goal of this work to match or beat the characteristic of CCM [19], and that meant doing a BC-based AEAD scheme.

All of that said, EAX can be viewed as having been derived from a generic-composition scheme we call EAX2, described in Section 7. Specifically, one instantiates the generic-composition scheme EAX2 with CTR mode (counter mode) and OMAC, and then collapses the two keys into one. If one does favor generic composition, EAX2 is a nice algorithm for it.

Why a conventional (two-pass) scheme? Having decided to give a BC-based AEAD scheme, why stick to a conventional (i.e., two-pass) one, avoiding the line of work that starts with [11]? This choice is difficult to justify for any reason beyond patent-avoidance. We have not attempted to do so.

Comparing CCM and EAX. In Figure 4 we compare some of the properties of CCM[19] and EAX. The count on block-cipher calls for EAX ignores key-setup costs. By the set Byte we mean $\{0,1\}^8$.

Other comments. Among the benefits of following what is basically an encrypt-then-authenticate approach is that invalid messages can be rejected with half the work of an authenticate-then-encrypt approach.

To obtain a MAC as efficient as the underlying PRF, use $\mathrm{MAC}_K(H) = \mathrm{Encrypt}_K^{0^n\ H}(\varepsilon)$.

In CCM [19] the tag-length parameter is authenticated. We have chosen not to do this because it is unnecessary to achieve our notion of security. Recall that the tag length, like the block-cipher itself, should be fixed and agreed-to, in an authenticated way, at the beginning of a session. It is a usage error to change parameters in the middle of a session. In light of this, authenticating the tag length has no known benefit.

Many applications won't care if their AE scheme is on-line—they know the length of the message in advance. Many applications won't care if they can pre-process a static header—perhaps the header is just a few blocks anyway. And so forth. Nothing we have done mandates the use of any novel feature of the provided scheme. The point is to *enable* it. The defining characteristic of a general-purpose mode of operation is that it *is* general purpose—we can't anticipate what will be of primary concern to the application, and so we need to try to anticipate the attributes that an application may find desirable and make sure that the algorithm itself doesn't stand in the way.

Finally, where does the name EAX come from? It stands for encrypt-then-authenticate-then-translate. Clearly we had problems with the spelling of "translate".

6 Intellectual Property Statement

The authors neither have, nor are of aware of, any patents or pending patents relevant to EAX. We do not intend to apply for any patents covering this technology. Our work for this note is hereby placed in the public domain. As far as we know, EAX is free and unencumbered for all uses.

7 EAX2 Algorithm

This section is not necessary to understand or implement EAX, but it is necessary for understanding the proof of EAX as well as the general approach taken for its design. That approach has been to first design a generic-composition scheme, EAX2, and then "collapse" to a single key for the particular case of CTR encryption and OMAC authentication.

EAX2 composition. Let $F\colon \mathsf{Key1} \times \{0,1\}^* \to \{0,1\}^n$ be a PRF, where $n \geq 2$. Let $\Pi = (\mathcal{E}, \mathcal{D})$ be an IV-based encryption scheme having key space $\mathsf{Key2}$ and IV space $\{0,1\}^n$. This means that $\mathcal{E}\colon \mathsf{Key2} \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^*$ and $\mathcal{D}\colon \mathsf{Key2} \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^*$ and $\mathsf{Key2}$ is a set of keys and for every $K \in \mathsf{Key2}$ and $\mathbf{N} \in \{0,1\}^n$ and $M \in \{0,1\}^*$, if $C = \mathcal{E}_K^{\mathbf{N}}(M)$ then $\mathcal{D}_K^{\mathbf{N}}(C) = M$. Let $\tau \leq n$ be a number. Now given F and Π and τ we define an AEAD scheme $\mathrm{EAX2}[\Pi, F, \tau] = (\mathrm{EAX2.Encrypt}, \mathrm{EAX2.Decrypt})$ as follows. Set $F_K^t(M) = F_K([t]_n \parallel M)$. Set $\mathsf{Key} = \mathsf{Key1} \times \mathsf{Key2}$. Then the encryption algorithm $\mathrm{EAX2.Encrypt}\colon \mathsf{Key} \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and the decryption algorithm $\mathrm{EAX2.Decrypt}\colon \mathsf{Key} \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \cup \{\textsc{Invalid}\}$ are defined in Figure 5 and the former is illustrated in Figure 6. $\mathrm{EAX2}[\Pi, F, \tau]$ is provably secure under natural assumptions about Π and F. See the full version of this paper.

Algorithm $\text{EAX2.Encrypt}^{N\,H}_{K1,K2}\,(M)$

$\mathbf{N} \leftarrow F^0_{K1}(N)$
$\mathbf{H} \leftarrow F^1_{K1}(H)$
$C \leftarrow \mathcal{E}^{\mathbf{N}}_{K2}(M)$
$\mathbf{C} \leftarrow F^2_{K1}(C)$
$\mathit{Tag} \leftarrow \mathbf{N} \oplus \mathbf{C} \oplus \mathbf{H}$
$T \leftarrow \mathit{Tag}\,[\text{first } \tau \text{ bits}]$
return $\mathcal{C} \leftarrow C \parallel T$

Algorithm $\text{EAX2.Decrypt}^{N\,H}_{K1,K2}\,(\mathcal{C})$

if $|\mathcal{C}| < \tau$ **then return** Invalid
Let $C \parallel T \leftarrow \mathcal{C}$ where $|T| = \tau$
$\mathbf{N} \leftarrow F^0_{K1}(N)$
$\mathbf{H} \leftarrow F^1_{K1}(H)$
$\mathbf{C} \leftarrow F^2_{K1}(C)$
$\mathit{Tag}' \leftarrow \mathbf{N} \oplus \mathbf{C} \oplus \mathbf{H}$
$T' \leftarrow \mathit{Tag}'\,[\text{first } \tau \text{ bits}]$
if $T \neq T'$ **then return** Invalid
$M \leftarrow \mathcal{D}^{\mathbf{N}}_{K2}(C)$
return M

Figure 5: The generic composition scheme $\text{EAX2}[\Pi, F, \tau]$. The scheme is build from a PRF $F\colon \mathsf{Key1} \times \{0,1\}^* \to \{0,1\}^n$ and an IV-based encryption scheme $\Pi = (\mathcal{E}, \mathcal{D})$ having key space Key2 and message space $\{0,1\}^*$.

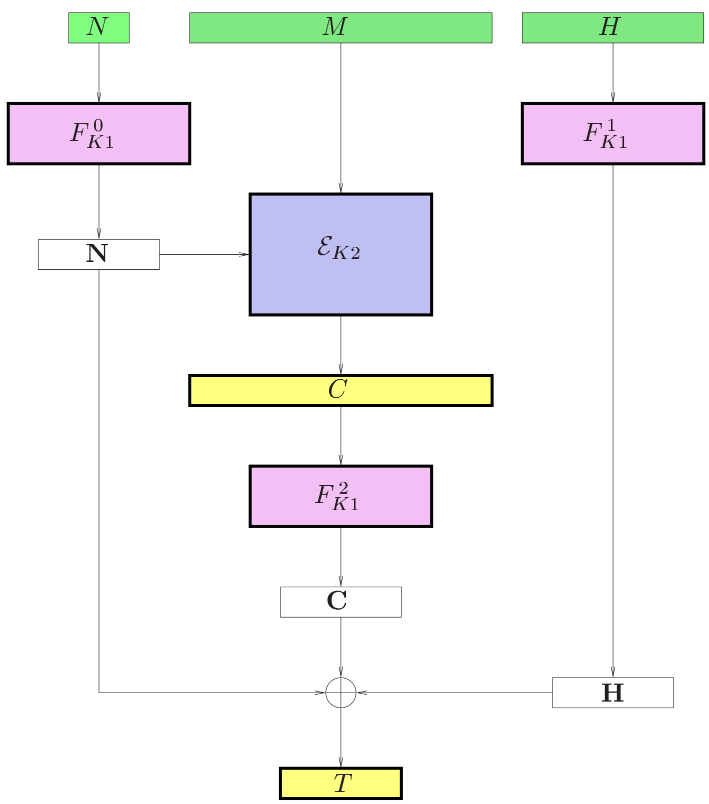

Figure 6: Encrypting under EAX2. The plaintext is M and the key is $(K1, K2)$ and the header is H. The ciphertext is $C \parallel T$. By F^i_K we mean the function where $F^i_K(M) = F_K([i]_n \parallel M)$.

EAX1 COMPOSITION. Let EAX1 be the single-key variant of EAX2 where one insists that $\mathsf{Key1} = \mathsf{Key2}$ and where one keys F, $\mathcal{E}$, and $\mathcal{D}$ with a single key $K \in \mathsf{Key} = \mathsf{Key1} = \mathsf{Key2}$. That is, one associates to F and Π, as above, the scheme $\mathrm{EAX1}[\Pi, F, \tau]$ that is defined as with EAX2 but where the key space is $\mathsf{Key} = \mathsf{Key1} = \mathsf{Key2}$ and the one key K keys everything. Notice that $\mathrm{EAX}[E, \tau] = \mathrm{EAX1}[\mathrm{CTR}[E], \mathrm{OMAC}[E], \tau]$. This is a useful way to look at EAX.

8 Security Theorem

EAX is a provably secure AEAD scheme if the underlying block cipher is a secure pseudorandom permutation (PRP). Proofs have been omitted from the current writeup. The full paper, to be released soon, will include them.

9 Acknowledgments

Mihir Bellare's work was funded by NSF grants CCR-0098123 and ANR-0129617, and by an IBM Faculty Partnership Development Award. Phil Rogaway's work was funded by NSF CCR-0208842 and a gift from CISCO Systems. David Wagner's work was funded by NSF CCR-0113941.

References

[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. *Proceedings of the 38th Symposium on Foundations of Computer Science*, IEEE, 1997. Available as `http://www-cse.ucsd.edu/users/mihir/papers/sym-enc.html`

[2] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. *Advances in Cryptology – CRYPTO '95*, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995. Available as and `http://www-cse.ucsd.edu/users/mihir/papers/xormacs.html`

[3] M. Bellare, O. Goldreich, and H. Krawczyk. Stateless evaluation of pseudorandom functions: Security beyond the birthday barrier. *Advances in Cryptology – CRYPTO '96*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996. Available as `http://www-cse.ucsd.edu/users/mihir/papers/otp.html`

[4] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* (JCSS), vol. 61, no. 3, pp. 362–399, Dec 2000. Available as `http://www-cse.ucsd.edu/users/mihir/papers/cbc.html`

[5] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in SSH: provably fixing the SSH binary packet protocol. *Proceedings of the 9th Annual Conference on Computer and Communications Security*, ACM, 2002. Available as `http://www-cse.ucsd.edu/users/mihir/papers/ssh.html`

[6] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000. Available as `http://www-cse.ucsd.edu/users/mihir/papers/oem.html`

[7] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient encryption. *Advances in Cryptology – ASIACRYPT '00*, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed., Springer-Verlag, 2000. Available as `http://www-cse.ucsd.edu/users/mihir/papers/ee.html`

[8] J. Black and P. Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. *Advances in Cryptology – CRYPTO '00*, Lecture Notes in Computer Science Vol. 1880, M. Bellare ed., Springer-Verlag, 2000. Available as http://www.cs.ucdavis.edu/~rogaway/papers/3k.html

[9] T. Iwata and K. Kurosawa. OMAC: One-key CBC MAC. *Fast Software Encryption '03*, Lecture Notes in Computer Science Vol. ?? , T. Johansson ed., Springer-Verlag, 2003. Also Cryptology ePrint archive Report 2002/180, http://eprint.iacr.org/2002/180

[10] T. Iwata and K. Kurosawa. Personal communications, January 2002.

[11] C. Jutla. Encryption modes with almost free message integrity. *Advances in Cryptology – EUROCRYPT '01*, Lecture Notes in Computer Science Vol. 2045 , B. Pfitzmann ed., Springer-Verlag, 2001. Also Cryptology ePrint archive Report 2000/039, http://eprint.iacr.org/2000/039/

[12] J. Katz and M. Yung. Unforgeable encryption and adaptively secure modes of operation. *Fast Software Encryption '00*, Lecture Notes in Computer Science Vol. 1978, B. Schneier ed., Springer-Verlag, 2000.

[13] J. Kilian and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences* (JCSS), vol. 61, no. 3, pp. 362–399, December 2000. Earlier version in CRYPTO 94.

[14] H. Krawczyk. The order of encryption and authentication for protecting communications (or: how Secure is SSL?). *Advances in Cryptology – CRYPTO '01*, Lecture Notes in Computer Science Vol. 2139, J. Kilian ed., Springer-Verlag, 2001. Also Cryptology ePrint archive Report 2001/045, http://eprint.iacr.org/2001/045

[15] M. Liskov, R. Rivest, and D. Wagner. Advances in Cryptology – CRYPTO '02, Lecture Notes in Computer Science, vol. 2442, pp. 31–46. Springer-Verlag, 2002. See www.cs.berkeley.edu/∼daw

[16] P. Rogaway. Authenticated-encryption with associated-data. *Proceedings of the 9th Annual Conference on Computer and Communications Security*, ACM, 2002. Available as http://www.cs.ucdavis.edu/~rogaway/papers/ad.html

[17] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. *Proceedings of the 8th Annual Conference on Computer and Communications Security*, ACM, 2001. Available as http://www.cs.ucdavis.edu/~rogaway/papers/ocb.htm

[18] P. Rogaway and D. Wagner. A critique of CCM. Manuscript, February 2003.

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

A Recommended API

Some important features of EAX can only be utilized if one accesses EAX functionality through an appropriate user interface. In this section we therefore put forward an API that permits (a) incremental encryption, (b) incremental decryption, (c) authenticity verification without ciphertext recovery, and (d) static headers with negligible per-message cost. Providing of these features results in an API that is a bit more elaborate than some programmers may want or need, so we also include some simpler, "all-in-one" calls.

```
/*
 * We provide two interfaces:
 * 1. A simple interface that does not support streaming data.
 * 2. An incremental interface that supports streaming data.
 * See below for documentation on both.
 */

/*******************************************************************
 * -- How to encrypt, the simplified interface --
 * First, call
 *     eax_init()
 * to setup the key and set the parameters.
 * Then, for each packet, call
 *    eax_encrypt()
 * When all done, call
 *   eax_zeroize()
 *******************************************************************
 * -- How to decrypt, the simplified interface --
 * First, call
 *   eax_init()
 * to setup the key and set the parameters.
 * Then, for each packet:
 *   eax_decrypt()
 * When all done, call
 *   eax_zeroize()
 * It is the caller's responsibility to check tag validity
 * by examining the return value of eax_decrypt().
 *******************************************************************/


/*******************************************************************
 * -- How to encrypt, incrementally --
 * First, call
 *     eax_init()
 * to setup the key and set the parameters.
 * Then, for each packet, call
 *    eax_provide_nonce()
 *    {eax_provide_header(), eax_compute_ciphertext()}*
 *    eax_compute_tag()
 * Here {x,y} means x or y, and z* means any number of iterations of z.
 * When all done, call
 *   eax_zeroize()
 *
 * Note that encryption can be done on the fly, and header and message data
 * may be provided in any order and in arbitrary chunks.
 *******************************************************************
 * -- How to decrypt, incrementally --
 * First, call
 *   eax_init()
 * to setup the key and set the parameters.
 * Then, for each packet:
```

```
 *   eax_provide_nonce()
 *   {eax_provide_header(), eax_provide_ciphertext()}*
 *   eax_check_tag()
 *   eax_compute_plaintext()    // only do this if tag was valid
 * When all done, call
 *   eax_zeroize()
 * Note that decryption may be done on the fly, and header and message data
 * may be provided in any order and in arbitrary chunks.
 * It is the caller's responsibility to check tag validity
 * by examining the return value of eax_check_tag().
 ******************************************************************/

typedef enum {AES128,AES192,AES256} block_cipher; /* "standard" ciphers */
typedef unsigned char byte;
typedef void eax_state;                      /* EAX context; opaque */


/*******************************************************************
 *   Calls common to incremental and non-incremental API
 ******************************************************************/

/*
 * eax_init
 *
 * Key and parameter setup to init a EAX context data structure.
 * If you don't know what to pass for t,E, use t=16, E=AES128.
 */
eax_state *
eax_init(
  byte* Key,       // The key, as a string.
  unsigned int t,  // The tag length, in bytes.
  block_cipher E   // Enumerated that indicates what cipher to use.
);

/*
 * eax_provide_header
 *
 * Supply a message header.  The header "grows" with each call
 * until a eax_provide_header() call is made that follows a
 * eax_encrypt(), eax_decrypt(), eax_provide_plaintext(),
 * eax_provide_ciphertext() or eax_compute_plaintext() call.
 * That starts reinitializes the header.
 */
int
eax_provide_header(
  eax_state *K,     // The EAX context.
  byte *H,          // The header (associated data) (possibly more to come)
  unsigned int h    // having h bytes
);

/*
 * eax_zeroize
 *
 * Session is over; destroy all key material and cleanup!
 */
void
eax_zeroize(
    eax_state *K       // The EAX context to remove
);
```

```
/*******************************************************************
 *  All-in-one, non-incremental interface
 *******************************************************************/

/*
 * eax_encrypt
 *
 * Encrypt the given message with the given key, nonce and header.
 * Specify the header (if nonempty) with eax_provide_header().
 */
int
eax_encrypt(
  eax_state *K,       // The caller provides the EAX context,
  byte* N,            // the nonce and
  unsigned int n,     // its length (in bytes), and
  byte* M,            // the plaintext and
  unsigned int m,     // its length (in bytes).
  byte* C,            // The m-byte ciphertext
  byte* T             // and the tag T are returned.
);


/*
 * eax_decrypt()
 *
 * Decrypt the given ciphertext with the given key, nonce and header.
 * Specify the header (if nonempty) with eax_provide_header().
 * Returns 1 for a valid ciphertext, 0 for an invalid ciphertext.
 */
int
eax_decrypt(
  eax_state *K,       // The caller provides the EAX context,
  byte* N,            // the nonce and
  unsigned int n,     // its length (in bytes), and
  byte* C,            // the ciphertext and
  unsigned int c,     // its length (in bytes), and the
  byte* T,            // tag.
  byte* P             // If valid, return the c-byte plaintext.
);


/*******************************************************************
 *      Incremental interface
 *******************************************************************/

/*
 * eax_provide_nonce
 *
 * Provide a nonce.  For encryption, do this before calling
 * eax_compute_ciphertext() and eax_compute_tag();
 * for decryption, do this before calling
 * eax_provide_ciphertext(), eax_check_tag, or eax_compute_plaintext().
 */
int
eax_provide_nonce(
  eax_state *K,    // The EAX context,
  byte* N,         // the nonce, and
```

```
  unsigned int n   // the length of the nonce (in bytes).
);


/*
 * eax_compute_ciphertext
 *
 * Encrypt a message or a part of a message.
 * The nonce needs already to have been
 * specified by a call to eax_provide_nonce().
 */
int
eax_compute_ciphertext(  // Encrypt (part of) a message
  eax_state *K,    // Given a EAX context K
  byte *M,         // and a message M  (possibly more to come)
  unsigned int m,  // having m bytes.
  byte *C          // Return a ciphertext body C also having m bytes.
);


/*
 * eax_compute_tag
 *
 * Message and header finished: compute the authentication tag that is a part
 * of the complete ciphertext.
 */
int
eax_compute_tag(
  eax_state *K,       // Given a EAX context
  byte *T             // compute the tag T for it.
);


/*
 * eax_provide_ciphertext
 *
 * Supply the ciphertext, or the next piece of ciphertext.
 * This is used to check for the subsequent authenticity check eax_check_tag().
 */
int
eax_provide_ciphertext(
  eax_state *K,       // Given a EAX context
  byte *C,            // and a ciphertext C (possibly more to come)
  unsigned int c      // having c bytes.
);


/*
 * eax_check_tag
 *
 * The nonce, ciphertext and header have all been fully provided; check if
 * they are valid for the given tag.
 * Returns 1 for a valid ciphertext, 0 for an invalid ciphertext
 * (in which case plaintext/ciphertext might be zeroized as well).
 */
int
eax_check_tag(
  eax_state *K,       // Given a EAX context and
  byte *T             // the tag that accompanied the ciphertext.
```

```
);


/*
 * eax_compute_plaintext
 *
 * Recover the plaintext from the provided ciphertext.
 * A call to eax_provide_nonce() needs to precede this call.
 * The caller is responsible for separately checking if the ciphertext is valid.
 * Normally this would be done before computing the plaintext with
 * eax_compute_plaintext().
 */
int
eax_compute_plaintext(
    eax_state *K,   // Given a EAX context
    byte *C,        // and a ciphertext C (possibly more to come)
    unsigned int c, // having c bytes,
    byte *M         // return the corresponding c bytes of plaintext.
);
```