AEZ v1: Authenticated-Encryption by Enciphering

Viet Tung Hoang
UC San Diego
vth005@eng.ucsd.edu

Ted Krovetz
Sacramento State
ted@krovetz.net

Phillip Rogaway
UC Davis
rogaway@cs.ucdavis.edu

March 15, 2014

The named authors are both designers and submitters.

Abstract

AEZ encrypts by appending to the plaintext a fixed authentication block and then enciphering the resulting string with an arbitrary-input-length blockcipher, this tweaked by the nonce and AD. The approach results in strong security and usability properties, including nonce-reuse security, automatic exploitation of decryption-verified redundancy, and arbitrary, user-selectable length expansion. AEZ is parallelizable and its computational cost is roughly 1.8 times that of AES-CTR. On recent Intel processors, AEZ runs at about 1.4 cpb on 2 KB messages.

The latest version of this document, and any other related material, can be found on the AEZ homepage: http://www.cs.ucdavis.edu/~rogaway/aez

Contents

0 Introduction

This document describes AEZ, which is both an enciphering scheme and an authenticated-encryption scheme. Before specifying it we provide a brief overview.

Authenticated encryption by enciphering. When we speak of an *enciphering scheme* we mean an object that is like a conventional blockcipher except that the plaintext's length is arbitrary and variable, and, additionally, there's a tweak. Regarding AEZ in this way, enciphering maps a key K, plaintext X, and tweak T to a ciphertext $Y = \text{Encipher}(K, T, X)$ having the same length as X. Going backwards, one can recover $X = \text{Decipher}(K, T, Y)$. The security property we seek is that of a tweakable, strong-PRP (pseudorandom permutation): for a random key K it should be hard to distinguish oracles $(\text{Encipher}(K, \cdot, \cdot), \text{Decipher}(K, \cdot, \cdot))$ from oracles $(\pi(\cdot, \cdot), \pi^{-1}(\cdot, \cdot))$ that realize a family of independent, uniformly random permutations and their inverse.

When we instead regard AEZ as an *authenticated-encryption* (AE) *scheme*, encryption maps key K, plaintext M, public nonce N (also called a "public message number"), associated data AD, and an authenticator length ABYTES to a ciphertext $C = \text{Encrypt}(K, N, AD, M)$ that is ABYTES bytes longer than M. Calling $\text{Decrypt}(K, N, AD, C)$ returns M or else an indication of invalidity. The security properties we seek is that of a *robust* authenticated-encryption scheme, a new and very strong notion that implies protection of the privacy and authenticity of M and the authenticity of N and AD, and must do so to the maximal extent possible even if nonces get reused ("misuse resistance" [30]), authenticator lengths are short, and, on decryption, invalid plaintexts might get prematurely released.

Why speak of enciphering schemes when CAESAR is a competition for AE schemes? Because an enciphering scheme of the form described determines an AE scheme by *encipher-to-AE conversion*. And the AE scheme one gets in this manner has attractive security and usability properties.

Encipher-to-AE conversion works like this. To encrypt, encipher a string X that encodes M and a block of ABYTES zero bytes using a tweak T that encodes N, AD, and the scheme's parameters. Decryption works by deciphering the presented string (again using the tweak determined by N and AD) and verifying the presence of the anticipated zero bytes. See Figure 1.

What are these "attractive security and usability properties" to which we allude? (1) If plaintexts are known *a priori* not to repeat, no nonce is needed for ensuring semantic security. (2) If there's redundancy in plaintexts whose presence is verified on decryption, this augments authenticity. (3) Any number of authenticator bytes can be selected. (4) Because of the last two properties, one can minimize length-expansion for low-energy or bandwidth-constrained applications. (5) If what's supposed to be a nonce should accidentally get repeated, the privacy loss is limited to revealing repetitions in (N, AD, M) tuples, and authenticity is not damaged at all. (6) If a decrypting party leaks which of multiple authenticity-checks fails (eg, by timing attacks or distinct error-codes), this won't compromise privacy or authenticity. (7) If a decrypting party leaks some or all of a putative plaintext that was supposed to be squelched by an authenticity check, this won't compromise privacy or authenticity.

The authors believe that the pleasant features just described would sometimes be worth a sizable computational price. The computational overhead for achieving these strong properties is relatively modest.

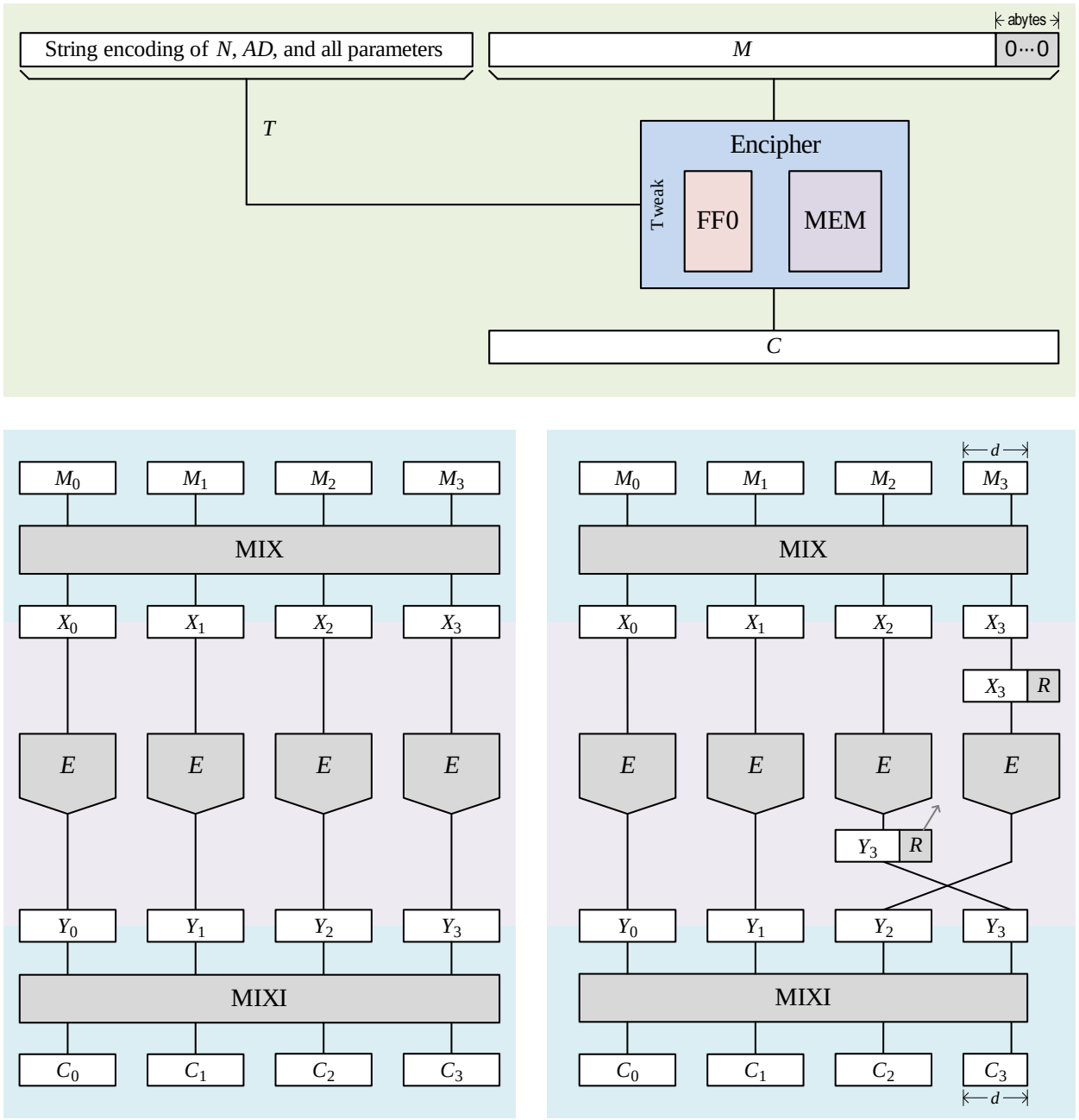

Figure 1: **High-level structure of AEZ**. After appending to the message a block of zeros we encipher it using a tweak that encodes the nonce, associated data, and parameters. The enciphering method depends on the length of what's being enciphered. The main mechanism, MEM, is shown beneath. The case for when the final block is full is on the left; the case for when it's a fragment is on the right. A layer of mixing is performed, then a layer of ECB-with-ciphertext-stealing, then a layer of inverse-mixing. For simplicity, keys are omitted from these illustrations and the incorporation of the tweak into MEM is not shown.

Realizing the enciphering. The heart of AEZ, then, is its enciphering scheme. The way AEZ enciphers depends on the length of what's being enciphered. When it's more than 16 bytes we follow what we call an "MEM" approach. The idea, derivative from Naor and Reingold (NR) [23, 24], involves a "non-cryptographic" mixing layer (it has only to satisfy a certain probabilistic property); a layer of ECB encryption (but now with ciphertext stealing); and a layer of inverse-mixing (using the same key as the initial mixing layer). See Figure 1. The mixing layers are realized from a MAC and an OCB-like offset sequence. The MAC is realized by a construction that looks like PMAC [5] but, in the spirit of MARVIN, PELICAN, and MT-MAC [8, 22, 32, 33], most of its computation uses four AES rounds instead of ten. The total cost comes out to be about 1.8 AES calls per block (four to mix; ten for ECB; four for the inverse-mix). In the end, encryption involves making two passes through the plaintext, both parallelizable and both using only constant memory (not counting exuding the ciphertext). We find that on a modern Intel processor, a preliminary AEZ implementation takes about 1.4 cycles per byte.

The name. The name "AEZ" is not exactly an acronym. The AE prefix is meant to suggest *authenticated encryption* and the overlapping EZ suffix is meant to suggest *easy*, in the sense of ease of correct use. The AES-like name is also a nod to the fact that AEZ is based on AES and can likewise be considered a species of blockcipher. Finally, the name can be used to identify individuals who can't distinguish an S from a Z.

1 Specification

1.1 Notation

Numbers and strings. For integers i, j with $i < j$, let $[i..j]$ be the set of integers $\{i, i+1, \ldots, j\}$. Strings are finite sequences of bits. The length of a string X, written $|X|$, is the number of bits it contains. The empty string ε is the string of length zero. Concatenation of strings A and B is denoted AB or $A \parallel B$. By 0^n we mean the string of n zero bits. If $\mathcal{S}$ is a set of strings then $\mathcal{S}^*$ is all strings, including ε, formed by concatenating elements of $\mathcal{S}$. If $|S| = n$ and $1 \le i \le n$ then $S(i)$ is the ith bit of S (indexing from the left starting at 1), $\mathrm{msb}(S) = S(1)$, and $\mathrm{lsb}(S) = S(n)$; and if $1 \le i \le j \le n$ then $S(i..j)$ is $S(i) \cdots S(j)$. The bitwise xor of equal-length strings A and B is denoted $A \oplus B$. By $A \oplus (B \parallel 0^*)$ we mean $A \oplus B0^p$ where $p = |A| - |B|$. By $A \oplus (0^* \parallel B)$ we mean $A \oplus 0^p B$ where $p = |A| - |B|$. Define $S \parallel 10^*$ as $S \parallel 10^p$ for the smallest p (possibly 0) where 128 divides $|S| + p + 1$. Parse a construction like $A \parallel B \parallel 10^*$ left-to-right, so this is $(A \parallel B) \parallel 10^*$.

A byte is string of length eight. The set of all bytes is denoted $\textsc{Byte}$. A byte string is an element of $\textsc{Byte}^*$. Let $\textsc{Byte}^{\le \ell}$ be the set of byte strings of length at most ℓ. The byte length of $X \in \textsc{Byte}^*$ is $\|X\| = |X|/8$. When $X \in \textsc{Byte}^*$ and $1 \le i \le j \le \|X\|$ then $X[i]$ is its ith byte (indexing from 1) and $X[i..j]$ is the substring of X that runs from its ith to jth byte (inclusive).

A block is 128 bits and a block string is a sequence of blocks. By $S[\![i]\!]$ we mean the ith block from the block string S, with indexing starting at zero. Note the different indexing convention, designed so that an $(11 \cdot 128)$-bit string *Klong* is $\mathit{Klong}[\![0]\!] \parallel \cdots \parallel \mathit{Klong}[\![10]\!]$.

For any $a \ge 0$ and $t \ge 1$ let $[a]^t$ be the t-byte string that encodes a in t bytes, assuming $a < 2^{8t}$. Here numbers are written in the customary way, most significant bit first. Similarly, $(a)_t$ is the t-bit string that encodes a, assuming $0 \le a < 2^t$. A value in typewriter font used as a string means the corresponding ASCII string: $\texttt{ABa} = [65]^1[66]^1[97]^1$.

If M is a string we write $(M_1, \ldots, M_m) \leftarrow M$ to indicate that m and $M_1, \ldots, M_m$ should be assigned the unique values such that $M_1 \cdots M_m = M$ and $|M_1| = \cdots = |M_{m-1}| = 128$ and $|M_m| \le 128$ with $M_m = \varepsilon$ only when $M = \varepsilon$. Define $(M_0, \ldots, M_m) \leftarrow M$ analogously (all blocks are full except possibly the last, which may be empty only when M itself is).

For $X \in \{0,1\}^{128}$ let $2 \bullet X = (X \ll 1) \oplus [135 \cdot \mathrm{msb}(X)]^{16}$. Let $0 \bullet X = [0]^{16}$, $1 \bullet X = X$, $(2n) \bullet X = 2 \bullet (n \bullet X)$, and $(2n+1) \bullet X = (2n \bullet X) \oplus X$.

AES. We assume familiarity with AES. We write $E_K(X) = E(K, X)$ for AES encipherment of 128-bit plaintext X using 128-bit key K. By $E_K^{-1}(Y)$ we denote the unique X for which $E_K(X) = Y$.

Given $K \in \{0,1\}^{128}$ let $\mathrm{MakeAESsubkeys}(K) \in (\{0,1\}^{128})^{11}$ be the string concatenating all 11 subkeys from the AES-128 key schedule. If $\mathit{Klist} = K_0 K_1 \cdots K_t$ is a block string, $|K_i| = 128$ and $t \ge 1$, then we write $E_{\mathit{Klist}}(X)$ for the function that enciphers X with t rounds of the AES round function, round i using subkey K_i, the last round omitting `MixColumns`, and the first round proceeded by an initial xor of K_0. Thus $E_K(X) = E_{\mathit{Klist}}(X)$ where $\mathit{Klist} = \mathrm{MakeAESsubkeys}(K)$. We write $E_{\mathit{Klist}}^{-1}(Y)$ for the unique X such that $E_{\mathit{Klist}}(X) = Y$. In discussions we write AES4 for the four-round version of AES, by which we mean E_K with $|K| = 5 \cdot 128$.

symbol	comments
M	Plaintext. $M \in \text{BYTE}^*$
C	Ciphertext. $C \in \text{BYTE}^*$
K	Key. $K \in \text{BYTE}^*$. Keys that aren't 16 bytes are first processed into 16-bytes
N	Nonce (aka: public sequence number). $N \in \text{BYTE}^{\leq 32}$. $\|N\| \leq 12$ most efficient
AD	Associated data. $AD \in \text{BYTE}^*$. Users should use the empty string if they don't need the AD
ABYTES	Authenticator length. ABYTES $\in [0..32]$. Default is 16. C will be ABYTES longer than M
EXTNS	Extensions directive. EXTNS $\in \text{BYTE}^3$. Default is $[0]^3$. Other values direct pre/post processing

Figure 2: **Arguments and parameters**. The first five values are arguments to our Encrypt( ) or Decrypt( ) routines. The next two values are parameters. The current document only specifies the behavior of AEZ when EXTNS = $[0]^3$. Other values will direct the invocation of integrated extensions.

1.2 Parameters

We'll take *parameter* to mean "a value on which AEZ encryption depends that we are expecting, independent of any particular API, to be held constant throughout some long-lived context." Thus we will not regard KEYBYTES as an AEZ parameter (we permit keys of any length), nor NPUBBYTES (we permit nonces to have varying lengths, even within a session). While these two values are omitted from the CAESAR-specified API, they could be specified in a different API. With this understanding, we will regard AEZ as having two parameters (and even those could just as well be considered as conventional arguments). See Figure 2.

- The *authenticator length*, ABYTES, quantifies the authenticity protection provided. It also determines how much longer a ciphertext is than its plaintext. The possible values of ABYTES are integers between 0 and 32 (inclusive). Values in excess of 16 are not claimed to provide additional security. While we call ABYTES a parameter, we do not insist that it be held constant throughout a session; an implementation is free to vary this value with each message encrypted. Still, we expect that most users will fix ABYTES for the duration of a session.
- The *extensions directive*, EXTNS, unlocks capabilities that have traditionally been seen as beyond-scope of an encryption scheme's functionality. These include: secret nonces (secret message numbers); plaintext-length obfuscation (via a specified padding regime); and encoding ciphertexts into a prescribed alphabet. These extension can be realized by a wrapper that keylessly transforms a plaintext, AEZ encrypts it, then keylessly transforms the result. A document defining these extensions, and how they are encoded by EXTNS, will be released later.

AEZ parameters have defaults; they are ABYTES = 16 and EXTNS = $[0]^3$. The only named parameter set, `aez`, uses these defaults. A conforming AEZ implementation is free to select defaults different from the ones given. It is also free to let the user select ABYTES and/or EXTNS through the argument list of procedures and to let these values vary across calls. In any context where the key length or nonce length are *required* to be fixed, we will select byte lengths for these of KEYBYTES = 16 and NPUBBYTES = 12.

Some readers may find fault with calling EXTNS an AEZ parameter but failing to specify at this time AEZ's behavior when it takes on a non-default value. The decision is part of a strategy to meld baseline AE functionality with a variety of additional features. A reader who persists in finding fault with the outlined approach is welcome to regard EXTNS as the constant $[0]^3$.

1.3 Pseudocode

Encryption and decryption. See Figure 3. To encrypt a string M we augment it with an *authenticator*—a block of ABYTES zero bytes—and encipher the resulting string, tweaking this enciphering scheme with a tweak formed from AD, N, and the parameters. These are encoded in a manner that enhances the efficiency of their processing (in particular, AD always starts with the second block and ends on a block boundary, and the nonce is packed into the first block as long as this is possible). Next we encipher the augmented message. To decrypt a ciphertext C we reverse the process, verifying the presence of the all-zero authenticator.

For the users' convenience, keys of any length are allowed. Using procedure Extract, they are first processed into 16-byte strings using an almost-universal hash function with a fixed but "random" key, an approach rooted in the leftover hash lemma [2, 10, 14]. The Extract algorithm is based on CMAC and NIST recommendation SP 800-56C [7]. Keys of 128 bits are processed more efficiently than other keys.

Alternative processing is performed at lines 104 and 113 if the message M is empty. In this case we do need not to encipher anything; the user is only requesting a message-authentication service. This saves some time when AEZ is used as a MAC. The MAC we use to satisfy the user's request is a PRF we call it AMAC. Besides taking in the key and the string that is being authenticated, AMAC also takes in a number $i \in [0..4]$, which is regarded as part of the domain of the PRF. The argument is used to conceptually provide a variety of MACS, each as efficient as the other. We will meet AMAC again; it is used for multiple purposes within AEZ.

Enciphering and deciphering. Messages are enciphered by one of four different methods. Dispatch occurs in algorithm Encipher of Figure 3. Strings of length 0 or 16 bytes are handled by Encipher itself. Strings of 1–15 bytes are enciphered using the Feistel-based method FF0, realized in algorithm EncipherFF0. Strings of 17 bytes or more are enciphered using a method we call MEM, realized in the algorithm EncipherMEM of Figure 4. In all of these routines, when encountering a key derived from K—any of $Kecb$, $Kff0$, $Kone$, $Kmac$, $Kmac'$, $Khash_i$, or K_i—the named key is implicitly defined from K using the procedure MakeSubkeyVectors of Figure 7.

Roughly following FFX [4, 11], algorithm EncipherFF0 uses ten rounds of a balanced Feistel network. (More rounds are used for strings shorter than three bytes. Specifically, we use 24 rounds for one-byte strings, and 16 rounds for two-byte strings.) The round function is based on AES. We use the four-round version of it for this purpose. This is implicit in the pseudocode, embedded in the fact that $Kff0$ is a five-block key. Another novel feature of EncipherFF0 compared to FFX is the swapping of a fixed pair of points when a key-dependent, tweak-dependent, length-dependent pseudorandom bit comes out to be 1. The same trick, without the tweak or length dependency, has been used before [26] to address the well-known fact that Feistel can only generate even permutations [16].

```
100  algorithm Encrypt(Key, N, AD, M)                    // AEZ authenticated encryption
101  K ← Extract(Key)
102  X ← M ‖ [0]^ABYTES
103  T ← Format(N, AD)
104  if M = ε then return AMAC(K, T, 4)[1..ABYTES]
105  C ← Encipher(K, T, X)
106  return C

110  algorithm Decrypt(Key, N, AD, C)                    // AEZ authenticated decryption
111  K ← Extract(Key)
112  T ← Format(N, AD)
113  if ‖C‖ = ABYTES then return (C = AMAC(K, T, 4)[1..ABYTES])
114  X ← Decipher(K, T, C)
115  M ‖ Z ← X where ‖Z‖ = ABYTES
116  if (Z ≠ [0]^ABYTES) then return ⊥
117  return M

120  algorithm Format(N, AD)                              // Encode inputs and parameters
121  if ‖N‖ ≤ 11 then return 00 ‖ (ABYTES)_6 ‖ EXTNS ‖ N ‖ 10* ‖ AD
122  if ‖N‖ = 12 then return 01 ‖ (ABYTES)_6 ‖ EXTNS ‖ N ‖ AD
123  if ‖N‖ ≥ 13 then return 10 ‖ (ABYTES)_6 ‖ EXTNS ‖ N[1..12] ‖ AD ‖ 10* ‖ N[13..‖N‖] ‖ [‖N‖]^1

130  algorithm Extract(Key)                               // Convert key into 128 bits
131  for i ← 1 to 4 do CONST_i ← E(AEZ-Constant-AEZ, [i]^16) od
132  if ‖K‖ = 16 then return K ← Key ⊕ CONST_1
133  (X_1, ..., X_m) ← Key;  K ← [0]^16
134  if ‖X_m‖ = 16 then X_m ← X_m ⊕ CONST_2 else X_m ← (X_m ‖ 10*) ⊕ CONST_3
135  for i ← 1 to m do K ← E(CONST_4, K ⊕ X_i) od
136  return K

200  algorithm Encipher(K, T, X)                          // AEZ enciphering
201  if ‖X‖ = 0 then return ε
202  else if ‖X‖ ≤ 15 then return EncipherFF0(K, T, X)
203  if ‖X‖ = 16 then Δ ← AMAC(K, T, 3);  return E_Kone(X ⊕ Δ) ⊕ Δ
204  else EncipherMEM(K, T, X)

210  algorithm EncipherFF0(K, T, M)                       // FF0 (1 ≤ ‖M‖ ≤ 15)
211  Δ ← AMAC(K, T, 2)
212  m ← |M|;  A ← M(1 .. m/2);  B ← M(m/2 + 1 .. m)
213  if m = 8 then k ← 24 else if m = 16 then k ← 16 else k ← 10
214  for i ← 1 to k do
215       B′ ← A ⊕ (E_Kff0([i]^4 ‖ B ‖ 10* ⊕ Δ))(1..m/2);  A ← B;  B ← B′  od
216  C ← A ‖ B
217  if Δ(m/8) = 1 and (C = 0^m or C = 1^m) then C ← C ⊕ 1^m
218  return C
```

Figure 3: **AEZ definition: Encrypt, Decrypt, Format, Extract, Encipher, EncipherFF0.** We encrypt M by appending an authenticator $[0]^{\text{ABYTES}}$ and then enciphering using a tweak that encodes AD, N, and the parameters. Deciphering checks for the authenticator. Support routines format the tweak and turn the user's key into 128 bits. The Encipher routine is responsible for length-dependent dispatch. It distinguishes $\|M\|$ being 0, being between 1 and 15, being 16, and being 17 or more.

```
220  algorithm EncipherMEM(K, T, M)                                  // MEM (‖M‖ ≥ 17)
221  (M_0, ..., M_m) ← M;  d ← ‖M_m‖
222  Δ ← AMAC(K, T, 0);  M_0 ← M_0 ⊕ Δ
223  X_0 ← AMAC(K, M_0···M_m, 1);  Y_0 ← E_Kecb(X_0)
224  for i ← 1 to m − 1 do
225       X_i ← M_i ⊕ X_0 ⊕ K_i;  Y_i ← E_Kecb(X_i);  C_i ← Y_i ⊕ Y_0 ⊕ K_i od
226  if d = 16 then
227       X_m ← M_m ⊕ X_0 ⊕ K_m;  Y_m ← E_Kecb(X_m);  C_m ← Y_m ⊕ Y_0 ⊕ K_m
228       C_0 ← E^{-1}(Kmac_1, Y_0) ⊕ AHash(K, C_1···C_m)
229  else
230       Y_m ← Y_{m−1}[1..d];  R ← Y_{m−1}[d + 1..16];  X_m ← M_m ⊕ X_0[1..d] ⊕ K_m[1..d]
231       Y_{m−1} ← E_Kecb(X_m ‖ R);  if m > 1 then C_{m−1} ← Y_{m−1} ⊕ Y_0 ⊕ K_{m−1} fi
232       C_m ← Y_m ⊕ Y_0[1..d] ⊕ K_m[1..d];  C_0 ← E^{-1}(Kmac'_1, Y_0) ⊕ AHash(K, C_1···C_m)
233  fi
234  C_0 ← C_0 ⊕ Δ
235  return C_0···C_m
```

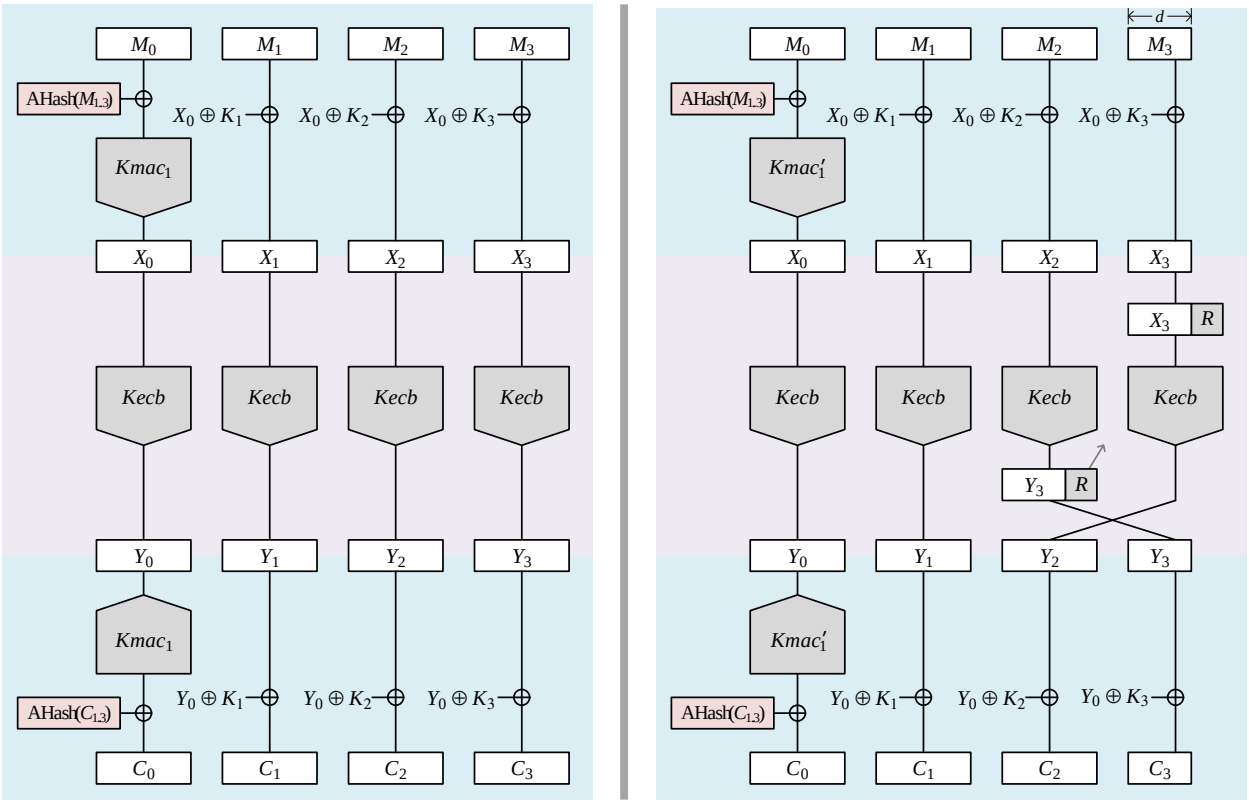

Figure 4: **AEZ definition: EncipherMEM.** The "heart" of AEZ is the MEM enciphering scheme. It is defined on top and illustrated below. The left side shows enciphering a string with a full final block; the right side shows the case of a fragmentary final block. The gray pentagons represent AES with the given key, the point representing the "forward" direction. In usage, block M_0 has already been offset by the image Δ of the PRF applied to the tweak, zero bytes have been added to the end of the underlying plaintext, and C_0 will be offset by Δ before the final string is output.

```
240  algorithm Decipher(K, T, C)                                   // AEZ enciphering
241  if ‖C‖ = 0 then return ε
242  else if ‖C‖ ≤ 15 then return DecipherFF0(K, T, C)
243  if ‖X‖ = 16 then Δ ← AMAC(K, T, 3); return E^{-1}_{Kone}(C ⊕ Δ) ⊕ Δ
244  else DecipherMEM(K, T, C)
```

```
250  algorithm DecipherFF0(K, T, C)                                // FF0 (1 ≤ ‖C‖ ≤ 15)
251  Δ ← AMAC(K, T, 2); m ← |C|
252  if Δ(m/8) = 1 and (C = 0^m or C = 1^m) then C ← C ⊕ 1^m
253  B ← C(1 .. m/2); A ← C(m/2 + 1 .. m)
254  if m = 8 then k ← 24 else if m = 16 then k ← 16 else k ← 10
255  for i ← r downto 1 do
256       B′ ← A ⊕ (E_{Kff0}([i]^4 ‖ B ‖ 10* ⊕ Δ))(1..m/2); A ← B; B ← B′ od
257  M ← B ‖ A
258  return M
```

```
260  algorithm DecipherMEM(K, T, C)                                // MEM (‖C‖ ≥ 17)
261  (C_0, ..., C_m) ← C; d ← ‖C_m‖
262  Δ ← AMAC(K, T, 0); C_0 ← C_0 ⊕ Δ
263  Y_0 ← AMAC(K, C_0···C_m, 1); X_0 ← E^{-1}_{Kecb}(Y_0)
264  for i ← 1 to m − 1 do
265       Y_i ← C_i ⊕ Y_0 ⊕ K_i; X_i ← E^{-1}_{Kecb}(Y_i); M_i ← X_i ⊕ X_0 ⊕ K_i od
266  if d = 16 then
267       Y_m ← C_m ⊕ Y_0 ⊕ K_m; X_m ← E^{-1}_{Kecb}(Y_m); M_m ← X_m ⊕ X_0 ⊕ K_m
268       M_0 ← E^{-1}(Kmac_1, X_0) ⊕ AHash(K, M_1···M_m)
269  else
270       X_m ← X_{m−1}[1..d]; R ← X_{m−1}[d + 1..16]; X_m ← M_m ⊕ X_0[1..d] ⊕ K_m[1..d]
271       X_{m−1} ← E^{-1}_{Kecb}(Y_m ‖ R); if m > 1 then M_{m−1} ← X_{m−1} ⊕ X_0 ⊕ K_{m−1} fi
272       M_m ← X_m ⊕ X_0[1..d] ⊕ K_m[1..d]; M_0 ← E^{-1}(Kmac′_1, X_0) ⊕ AHash(K, M_1···M_m)
273  fi
274  M_0 ← M_0 ⊕ Δ
275  return M_0···M_m
```

Figure 5: **AEZ deciphering: Decipher, DecipherFF0, and DecipherMEM.** These algorithms realize the inverses of Encipher, EncipherFF0, and EncipherMEM.

The EncipherMEM routine of Figure 4 roughly follows Naor and Reingold [23, 24]. The acronym MEM stands for Mix, ECB-with-ciphertext-stealing, Mix-inverse. The mixing function employs AMAC, already introduced, which is in turn based on the universal hash function we call AHash.

We define Decipher(K, T, Y) as the unique X such that Encipher$(K, T, X) = Y$. From the point of view of having a well-defined specification, this is all that one need say. We nonetheless provide an algorithmic description of Decipher, DecipherFF0, and DecipherMEM in Figure 5. Algorithm DecipherFF0 is like EncipherFF0 except that (i) it does the possible-swapping of 0^m and 1^m at the beginning instead of the end, (ii) it counts down from r to 1 instead of up from 1 to r, where r is the round number, and (iii) it performs an $(|m|/2)$-bit rotational shift before, and after the for-loop. Algorithm DecipherMEM is like EncipherMEM except that blockcipher calls E_{Kecb} are replaced by calls to the inverse blockcipher E^{-1}_{Kecb}.

```
300  algorithm AMAC(K, M, i)                                          // The PRF
301  (M_0, ..., M_m) ← M
302  if ‖M‖ < 16 then return E(Kmac'_i, M ‖ 10*)
303  else if ‖M‖ = 16 then return E(Kmac_i, M)
304  else if ‖M‖ mod 16 = 0 then return E(Kmac_i, M_0 ⊕ AHash(K, M_1···M_m))
305  else return E(Kmac'_i, M_0 ⊕ AHash(K, M_1···M_m))
```

```
310  algorithm AHash(K, M)                                            // AXU hash
311  (M_1, ..., M_m) ← M;  d ← ‖M_m‖;  Σ ← [0]^16
312  for j ← 1 to m − 1 do Σ ← Σ ⊕ E(Khash_j, M_j)  od
313  if d = 16 then Σ ← Σ ⊕ E(Khash_m, M_m) else Σ ← Σ ⊕ E(Khash_m, M_m ‖ 10*)
314  return Σ
```

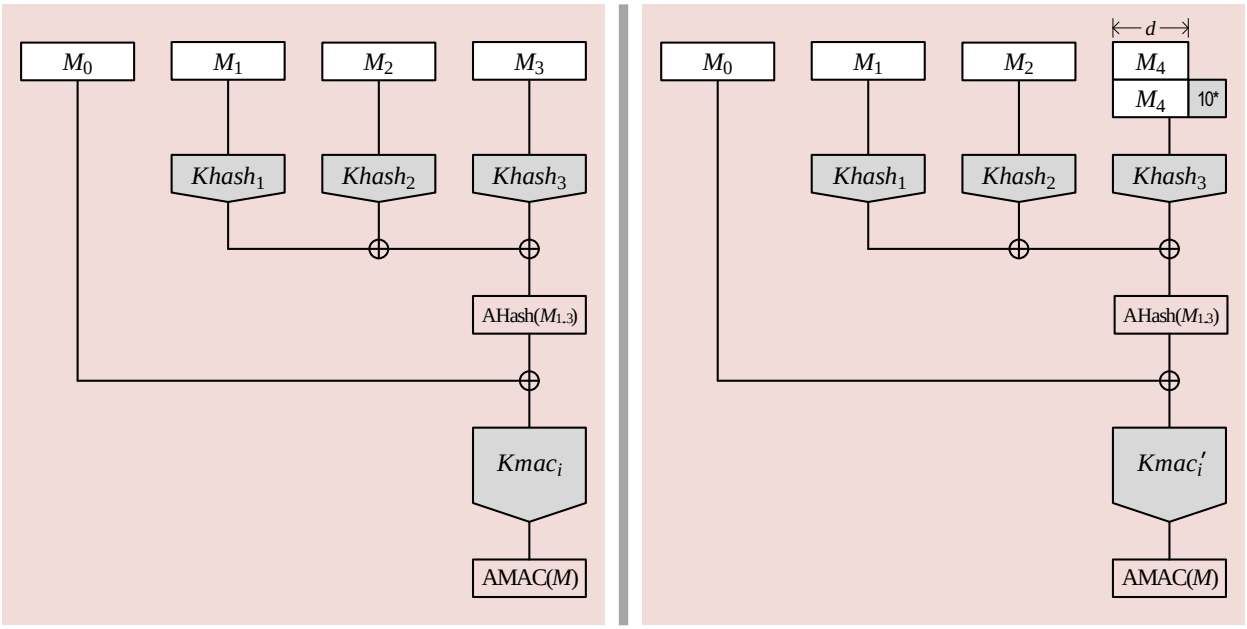

Figure 6: **Top: Definition of AMAC.** The domain point it operates on is pairs (M, i) where M is a string and $i \in [0..4]$ is a number. The MAC uses the hash function AHash, which operates on strings that are a positive multiple of 128 bits. **Bottom: Illustration of AMAC**. On the left: processing of four full-blocks. On the right: the final block is a fragment. The gray pentagons are four-round AES (top layer) and the full AES (the bottom layer) using the named key.

AMAC, AHash, and key derivation. Figure 6 defines the message authentication code AMAC. It is used multiple ways: to make the offset Δ that reflects the tweak of MEM; in the mixing function of MEM; to make the offset Δ that reflects the tweak of FF0; to make the offset Δ that reflects the tweak when enciphering one-block messages; and to make an efficient MAC when the plaintext length is zero. Each context employs its own tweak $i \in \{0, 1, 2, 3, 4\}$. AMAC is built in the Carter-Wegman tradition with a universal hash function, AHash, based on AES4. A similar constructions in the literature is MARVIN [32, 33], which itself borrows from PMAC and Pelican [5, 8].

See Figure 7 for a definition of how subkeys are manufactured from a single 128-bit key K. The approach is rooted in the XE and XEX construction of tweakable blockciphers [19, 29].

```
400  algorithm MakeSubkeyVectors(K)                                  // Subkey generation
401  Kecb ← Variant(K, 0,0,1, 10)
402  Kff0 ← Variant(K, 0,0,2, 4)
403  Kone ← Variant(K, 0,0,3, 10)
404  for i ← 0 to 4 do
405       Kmac_i ← Variant(K, 0,0,4 + i, 10)
406       Kmac'_i ← Variant(K, 0,0,9 + i, 10)  od
407  for i ∈ {1,2,...} do
408       K_i ← Variant(K, 2^⌈i/8⌉, (i − 1) mod 8, 0, 0)
409       Khash_i ← Variant(K, 2^⌈i/8⌉, (i − 1) mod 8, 0, 4)  od
```

```
410  algorithm Variant(K, j, i, ℓ, k)                                // Make subkey (i, j, ℓ)
411  Klong ← MakeAESsubkeys(K)
412  Kshort ← [0]^16 ‖ Klong[[2]] ‖ Klong[[5]] ‖ Klong[[8]] ‖ [0]^16   // In AEZ+: Kshort ← Klong
413  I ← E_K([0]^16);  J ← E_K([1]^16);  L ← E_K([2]^16)
414  Offset ← (j • J) ⊕ (i • I) ⊕ (ℓ • L)
415  if k = 0 then return Offset
416  if k = 4 then return Kshort ⊕ (Offset ‖ 0*)
417  if k = 10 then return Klong ⊕ (Offset ‖ 0*) ⊕ (0* ‖ Offset)
```

Figure 7: **Subkey generation.** The routine derives all needed subkeys from K. Procedure MakeSubkeyVectors is explicitly called; reference to a subkey implicitly employs these definitions.

1.4 Usage cap

We assert that a given key should act on at most 2^{52} bytes (4 petabytes); by that time, the user should rekey. For the purpose of this requirement, we say that, when encrypting (N, AD, M) with a given key K, AEZ is acting on $\|N\| + \|AD\| + \|M\| + 64$ bytes.

The above requirement stems from the existence of birthday attacks on AEZ; there *are* attacks on AEZ that use s blocks and then violate AEZ's security with advantage of about $s^2/2^{128}$. Retaining security thus requires keeping s to well under 2^{64} blocks. There is inherently some arbitrariness in deciding how to impose such a limit.

2 Security Goals

Nonce-reuse security. AEZ targets *nonce-reuse security*, as previously defined by Rogaway and Shrimpton under the name *misuse-resistant AE* [30]. In an MRAE scheme, repeating a nonce will violate privacy only insofar as repetitions of (N, AD, M) tuples will be revealed by ciphertexts, and it will not compromise authenticity at all. SIV [30] is the best-known MRAE scheme.

MRAE security implies automatic exploitation of unpredictability present in messages. In any context where messages are known to be distinct (eg, a sequence number is embedded within) or are extremely unlikely to collide (eg, a random session key is embedded within), use of a nonce is unnecessary. Its omission or reuse in such contexts is not misuse, but an appropriate way to encrypt.

Exploitation of domain-specific redundancy. In many contexts, plaintexts have a certain expected structure. This might arise because the message was produced in conformance with known constraints or by a particular protocol. If the user checks for anticipated structure and discards messages that fail to comply, the redundancy augments authenticity and lessens the need for the nominal redundancy (the all-zero authenticator) inserted before enciphering. It is a goal of AEZ to demonstrably exploit externally verified redundancy to augment authenticity, a folklore idea already known to work under one formalization of enciphering-based AE [3].

Releasing unverified plaintext. When decrypting, an *unverified plaintext* is a string that will be released if the ciphertext is deemed authentic, but is supposed to be quashed otherwise. While not definitionally mandated, AE schemes routinely compute such a thing. One form of encryption-scheme misuse is to release some or all of the unverified plaintext despite the ciphertext's invalidity. This might happen because of an incremental decryption API or a more traditional side-channel.

Contemporaneous work by Andreeva *et. al* gives definitions to formalize an AE scheme's security despite release of unverified plaintexts. Our own definitional approach is different; we formalize *robust* AE security, which incorporates the unverified-plaintext concern among its aspects.[1] In claiming robust AE security for AEZ the unverified plaintext is, of course, the value M computed at line 115. Achieving robust AE implies that no harm would come of returning $(M, \perp)$ instead of $\perp$ at line 116.

Per-message nonce-length / parameter authentication. No security problem results from employing nonces of varying lengths during a session, nor from changing the authenticator length ABYTES. Of course accessing such capabilities requires a competent API. But, algorithmically, every verified ciphertext can and should imply a consonant view (between the encrypting and decrypting parties) as to the current values of $|N|$, ABYTES, and EXTNS.

Optimal security for the plaintext expansion. Traditionally, AE security definitions "gave up" when the adversary forges. This means that, at least definitionally, it's OK for a scheme to fail catastrophically as soon as it fails. A consequence is that authentication tags need to be so long that forgeries almost never occur. Yet there are applications where an occasional forgery is

[1] We formerly used the terms *speculative plaintext* and *putative plaintext* to mean what Andreeva *et. al* call *unverified plaintext* [1]. Not wanting to pointlessly muddy this water, we now adopt the term from [1].

fine. For example, in some settings it is fine to have a 1-byte authenticator. While the adversary would have a 2^{-8} chance of forging a given message, we could still ensure that, say, a reasonable adversary won't have much more than a 2^{-80} chance to forge ten messages in a row.

AEZ permits short authentication tags, getting security as strong as possible given the length of the authenticator. This implies that we should use a new definition for AE, one that does not "give up" when a forgery occurs. It is described next.

Robust AE. Our new notion of AE captures that one is doing *as good a job as possible for a given value τ of plaintext expansion* ($\tau = 8 \cdot \text{ABYTES}$). We fold into this the notion of security in the face of leaking unverified plaintexts. The academic paper corresponding to the current submission will define and investigate our notion of *robust AE* (RAE). Here we sketch the idea.

To begin, we split the decryption functionality $\mathcal{D}$ into two pieces: algorithm $\tilde{\mathcal{D}}$ computes the unverified plaintext, while algorithm $\mathcal{V}$ indicates if it should be regarded as valid. Specifically, an RAE scheme is presented as a triple of deterministic algorithms $\Pi = (\mathcal{E}, \tilde{\mathcal{D}}, \mathcal{V})$, the *encryption* algorithm, the *unguarded-decryption* algorithm, and the *validity* algorithm. These work with associated sets $\mathcal{K}$, $\mathcal{N}$, and $\mathcal{AD}$ specifying the *key space*, the *nonce space*, and the *AD-space*. The encryption algorithm maps K, N, AD, M to a ciphertext $C = \mathcal{E}_K^{N,AD}(M)$. The unguarded-decryption algorithm maps K, N, AD, C to an unverified plaintext $M = \mathcal{D}_K^{N,AD}(C)$. The *ciphertext-validity* algorithm maps K, N, AD, C to a bit $v = \mathcal{V}_K^{N,AD}(C)$ indicating if the unverified plaintext $M = \mathcal{D}_K^{N,AD}(C)$ should be considered as valid (when $v = 1$) or invalid (when $v = 0$). The *decryption algorithm* $\mathcal{D} = [\tilde{\mathcal{D}}, \mathcal{V}]$ returns $\mathcal{D}_K^{N,AD}(C) = \tilde{\mathcal{D}}_K^{N,AD}(C)$ if $\mathcal{V}_K^{N,AD}(C) = 1$ and $\mathcal{D}_K^{N,AD}(C) = \perp$ otherwise. We assume that there's a number τ, the ciphertext expansion, such that $|C| = \tau + |M|$ when $C = \mathcal{E}_K^{N,AD}(M)$.

An adversary is dropped into one of two settings. Both provide two oracles, which we'll call Enc and Dec. In the *real* encryption setting we choose a key $K \xleftarrow{\$} \mathcal{K}$ uniformly at the beginning of the experiment. Then Enc, on input (N, AD, M), returns $\mathcal{E}_K^{N,AD}(M)$. The Dec oracle, on input (N, AD, C), returns $(\tilde{\mathcal{D}}_K^{N,AD}(C), \mathcal{V}_K^{N,AD}(C))$. Thus the Dec oracle, unlike proper decryption, is assumed to return the unverified plaintext even when the ciphertext is invalid.

The *ideal* setting works differently. First, for each (N, AD), we choose a uniform random injective function $\pi_{N,AD}(\cdot) \colon \{0,1\}^* \to \{0,1\}^*$ for which $|\pi_{N,AD}(M)| = |M| + \tau$. When the adversary asks $\text{Enc}(N, AD, M)$ we answer with $\pi_{N,AD}(M)$. This models encryption that's as good as possible among schemes with τ bits of expansion.

How do we answer $\text{Dec}(N, AD, C)$ queries? If there's an M for which $\pi_{N,AD}(M) = C$ then we answer $(M, 1)$. Otherwise we answer $(M, 0)$ for some *synthetic* plaintext M returned by a *simulator*, S, which is a probabilistic algorithm. It's provided N, AD, and C, and any saved state it wishes to maintain. It returns a string τ bits shorter than C. Saying that an AE scheme achieves RAE security means that there exists a (simple, efficient) simulator S for which no adversary can do well at distinguishing the real and ideal settings of the game just described.

Entropy extraction from arbitrary-length keys. It is a goal of AEZ to allow keys of arbitrary lengths, and to do a reasonably good job of entropy extraction when processing keys that are not already 128 bits. In particular, 128-bit keys are assumed to be uniformly random, but keys of other lengths should be sensibly processed to 128-bit ones. We assume that user keys are not specifically constructed so as to frustrate their transformation into 128-bit keys.

Provable security. AEZ has been developed with provable security strongly in mind. Just the same, we have not insisted on having provable security, based only on the assumption that AES is a strong PRP, cover all capabilities of the scheme. In particular, our provable-security results assume that plaintexts, after appending the authenticator, are at least 128 bits, and they assume that keys are uniformly random strings of 128 bits. Further assumptions are needed for some results.

Let us illustrate the kind of provable-security result we target. One shows that when the plaintext space and key space are as indicated above, the algorithm we call AEZ$^+$ achieves RAE security, up to the birthday bound, if blockcipher E is a strong-PRP. The difference between AEZ$^+$ and AEZ is this: for the first, line 412 is $\mathit{Kshort} \leftarrow \mathit{Klong}$. This means that the full AES, not AES4, will be used to construct the universal hash function AHash. Moving from AEZ$^+$ to AEZ decreases the construction's cost from about 3.0 AES calls per block to about 1.8 AES calls per block. Heuristic reasons, some further rooted in provable-security considerations, are offered to explain why the use of AES4 instead of AES is reasonable at this point in the algorithm.

Security non-goals. We have not tried to achieve security beyond the birthday bound. Like most modes based on a 128-bit blockcipher, there *are* attacks that succeed with high probability if the adversary can query AEZ with about 2^{64} in message, AD, and nonce material.

3 Security Analysis

An academic paper with the relevant security proofs for AEZ is in preparation. In the meantime, we summarize our provable-security results for AEZ.

In the analysis, we sometimes pretend that the subkeys for AES4 (excluding the XE offsets) are independent of other keys. In the implementation, to reduce context size, we steal the needed subkeys for AES4 from full AES, the likelihood of a problematic interaction seeming far-fetched.

Ciphertexts of at least one block. The security of AEZ can be proven under the assumption that AES is a strong PRP and AMAC is a PRF. This assumes that we are using either MEM or single-block enciphering. Equivalently, we are assuming $\|M\| + \text{ABYTES} \geq 16$ for each encryption query and $\|C\| \geq 16$ for each decryption query. We also assume that the underlying key is 16 bytes. With those provisos, robust-AE security can be proven for AEZ along the following lines.

- Due to the use of the XEX construction [19, 29] in realizing a tweakable blockcipher, AES calls with keys $\mathit{Kecb}, \mathit{Kone}, \mathit{Kmac}_i$, and Kmac'_i (for $i \in \{0, 1, 2, 3, 4\}$) are indistinguishable from AES calls with independent keys, up to a birthday-bound advantage.
- Ignoring the tweaking at lines 222 and 234, the MEM construction provably yields a length-preserving, strong PRP on $\text{BYTE}^{\geq 17}$, with birthday-bound distinguishing advantage. The proof only requires that AMAC resist non-adaptive queries. This observation facilitates justifying use of the same offset sequence K_i in AMAC and in producing the Mix-layer output the feeds the ECB-layer input of MEM.
- The tweak provided to EncipherMEM is incorporated by what can again be regarded as the XEX construction.

- AEZ enciphering on 128-bit strings also provably yields a tweakable, strong PRP on $\{0,1\}^{128}$, as the method employed here is again XEX-based.
- Once one has shown that the Encipher procedure of AEZ provides a tweakable, strong PRP then AEZ itself is a robust AE scheme. This is a generic result that asserts that encipher-to-AE conversion gives RAE security.

With security results that fall off with the birthday bound, it is natural to ask if there are corresponding attacks. It is easy to see that there are. As a simple example, let ABYTES = 16 and have an adversary repeatedly ask to encrypt a fixed message M with a fixed nonce N but using AD values that consist of two random blocks. A collision in ciphertexts will be found in about 2^{64} expected queries. Say it arose from AD values of $AD = AD[\![0]\!]AD[\![1]\!]$ and $AD' = AD'[\![0]\!]AD'[\![1]\!]$. Then test if one again gets a collision with M and N but with AD values of either $AD \,\|\, [0]^{16}$ or $AD' \,\|\, [0]^{16}$. If so, one almost certainly has a "real" encryption oracle.

Security of AMAC. The PRF security for AMAC can be heuristically justified by viewing AMAC as an approximation of a variant in which the keys $Khash_i$ are chosen uniformly and independently from $\{0,1\}^{128\cdot 4} \times \{0^{128}\}$. This variant of AMAC is a PRF due to the fact that four-round AES with independent, uniformly random subkeys is an AXU hash [17] and the fact that AMAC is constructed from the Carter-Wegman paradigm [6].

Alternatively, one can view AMAC as an approximation of an AES-based PMAC [5], in which all but the final blockcipher call have had the number of AES rounds reduced from 10 to 4, a heuristic employed in ALRED, MARVIN, and PELICAN [8, 9, 32, 33].

In particular, if we replace line 412 by $Kshort \leftarrow Klong$, the variant of AEZ we called AEZ$^+$, then we'd achieve provable RAE security under the sole assumption that AES is a good PRP. For in this setting AMAC becomes PMAC and all keys would be properly separated.

Ciphertexts of less than a block. The claim that EncipherFF0 gives a tweakable, strong PRP over $\text{BYTE}^{<16}$ is heuristically justified. Consider a collection of independent, ideal, k-round Feistel networks on $\{0,1\}^{2n}$; the round functions are all uniformly random and independent. The best attack known that distinguishes them from a family of independent, truly random even permutations, requires at least $2^{(k-4)n}$ plaintext/ciphertext pairs [25]. From our choice of the number of rounds, this attack needs at least 2^{72} plaintext/ciphertext pairs, and thus doesn't violate our up-to-the-birthday-bound security goal.

There are of course many provable-security results on balanced Feistel as well, but proven bounds for a fixed-round Feistel network operating on an m-bit string vanish at about $2^{m/2}$ queries, and we are looking at settings with m as small as 8.

Key processing. At present we view the entropy extraction procedure Extract as essentially heuristic, although some provable-security claims about it can be made [2, 10, 14]. The method is similar to CMAC and is based on NIST recommendation SP 800-56C [7].

4 Features

See Section 2 for security goals that are effectively features of AEZ and Figure 8 for a summarizing table. Below we enumerate additional features and restate some already mentioned one.

1) Strings of any byte length m can be encrypted into strings of any byte length m+ABYTES where $0 \le$ ABYTES ≤ 32. One achieves the maximal privacy and authenticity protection consistent with ABYTES. The value ABYTES is authenticated and may change as often as a user likes.
2) Nonces are optional (fix $N = \varepsilon$ if unused). If used, they can have any length from 0 to 32 bytes. If unused, one gets the strongest possible security notion in their absence.
3) Keys can have any length. A user may, for example, use a passphrase or DH ephemeral key. (Note: some features one might want for mapping a passphrase to a 128-bit key, like salting and an intentionally slow mapping to slow password guessing, are not provided.)
4) AEZ functions well as a stand-alone MAC and as a stand-alone enciphering scheme.
5) Verification of plaintext redundancy enhances authenticity, as we have already emphasized.
6) Short authenticators provide the security one would hope for. Our security notions don't "give up" when the adversary forges.
7) Release of unverified plaintext does not cause any problems for AEZ. This is part of the robust-AE definition.
8) The security properties we achieve mean that secret message numbers are easily accommodated within AEZ itself. This will be specified by an AEZ extension.
9) Further AEZ extensions will handle plaintext-length obfuscation, encoding into a target alphabet, vector-valued plaintexts and vector-valued AD. The extensions used are automatically authenticated with each message.
10) An encryption implementation can make one left-to-right, constant-memory pass over the input, and then a second left-to-right, constant-memory pass over the input, this time outputting the ciphertext online. Decryption can be similarly realized.
11) It is possible to accelerate the rejection of invalid ciphertexts by having decryption compute the final ciphertext block M_m prior to computing the remainder of the plaintext, $M_0 \cdots M_{m-1}$. This can make the rejection of invalid ciphertexts roughly as fast as AMAC itself, which runs in less than 0.3 cpb on a recent Intel machine.
12) AEZ is fully parallelizable in the processing of plaintext, ciphertext, and AD.
13) Static AD can be preprocessed so that one doesn't have to subsequently pay a per-message $|AD|$-dependent cost. (Note: realizing this benefit requires an API that decouples provisioning of the AD and provisioning of other inputs.)
14) Word alignment of the message and AD are not disrupted (for example, one never prepends a byte to the message or AD, and then processes it).
15) Nothing in the specification favors one endian convention over another. (Note: to keep the specification simple, AEZ is described in what amounts to a big-endian convention.)
16) No AEZ-related patents have been or will be requested.

A preliminary implementation of AEZ encrypts 2 KB messages at about 1.4 cpb on an Intel Haswell CPU.

Key length	**Arbitrary**. Keys shorter or longer than 128 bits are automatically processed into 128 bits first. This for user convenience; keys longer than 128 bits don't augment security.
Tag size	**0–32 bytes**. Here "tag size" means ABYTES; there is no actual tag. Expansion by 0 bytes corresponds to AEZ's use as a strong, tweakable, VIL blockcipher.
Truncation	**Yes**. While there is no tag to truncate, we interpret the intent as: for authenticity, it is possible to request an arbitrary number of bytes less than some maximum.
Sponge	**No**. The construction is not based on the sponge.
State Size	**448 bytes**. This is approximate; there are tradeoffs in deciding what values to maintain in the state and what to recompute.
Security	About $\mathbf{128 - 2\lg\sigma}$ bits of **privacy** and $\mathbf{\min(\tau,\ 128 - 2\lg\sigma)}$ bits of **authenticity**, where σ is the total number of blocks of plaintext, ciphertext, nonce, and AD queried, and $\tau = 8 \cdot \text{ABYTES}$. Regarded as lower bounds, these values depend on assumptions.
Nonce reuse	**Yes**. AEZ is secure against nonce-reuse in the strongest sense of the phrase [30].
Ciphertext misuse	**Yes**. It is fine to release unverified plaintext (a recovered but inauthentic plaintext): no security problems result. This is one aspect of our notion of a *robust* AE.
Proofs	Either: **Yes**, there are proofs, but then a heuristic optimization is applied to a provably-secure scheme to get a nice speedup; or **Yes**, there are proofs, but under a nonstandard assumption; or **No**, there are no proofs for AEZ itself, although the authors employ provable-security to motivate and justify design choices.
Parallelizable	**Yes**. Two passes must be made to encrypt or decrypt, but both are parallelizable. Processing of the AD is also parallelizable.
Incremental	**No**. Misuse-resistant schemes [30] can't be incremental. Use as a deterministic MAC is incremental with respect to block replacement or appending-on-the-right.
Online enc/dec	**No/No**. Misuse-resistant schemes [30] can't be online with respect to encryption or decryption. Processing the AD is online.
Performance	**sw/hw/lw**. Roughly 1.8× the cost of AES-CTR. Intended to do well where AES does, in software or hardware and on low-power devices where ciphertext length should be minimized. Not intended for processing extremely long messages.
Inverse	**Yes**. Decryption needs AES^{-1}, the inverse direction of the underlying blockcipher.
Side channels	**AES**. Implementable in time that depends only on the length of the input assuming that AES itself is so implemented.
Properties	▸ Can exploit arbitrary redundancy in messages for authenticity (just verify that the decrypted message is of the right form). ▸ Parameters are authenticated and may vary during a session. ▸ Nonces may vary in length, even within a session. ▸ Can use as a tweakable enciphering scheme (just set ABYTES = 0). ▸ Can use as an efficient, parallelizable MAC (encrypt the empty string). ▸ Can use to encipher very short strings, and to encrypt short strings with low expansion (eg, a 5-byte plaintext to an 8-byte ciphertext). ▸ AEZ extensions (to be specified separately) support secret nonces, plaintext-length obfuscation, and radix64url output encoding. ▸ Best-possible security for short authenticators. ▸ Achieves *robust* AE, a new and very strong security notion. ▸ Two passes. Both are online and use only $O(1)$ memory. ▸ Invalid ciphertexts can be rejected faster than a full decryption: about 0.4× the cost of AES-CTR. ▸ No patents.

Figure 8: **Table of properties for AEZ**. The choice of properties to list as the rows of this table is taken from slides prepared by Bart Preneel during a recent Dagstuhl workshop [27].

Advantages over GCM. AEZ has much stronger security properties than GCM. The later is not nonce-reuse secure, cannot safely generate short tags [12], and is not secure with respect to disclosure of unverified plaintext. GCM does not achieve the RAE security definition. AEZ avoids $\mathrm{GF}(2^{128})$ multiplies (apart from the finite-field "doubling" that it uses).

A closer match to AEZ in terms of high-level aims is SIV, which is likewise nonce-reuse secure [30]. But SIV has to output 128-bits more than its input; it cannot exploit verified redundancy in ciphertexts; and it is not parallelizable (although the last issue could easily be fixed).

5 Design Rationale

Enciphering-based AE. An old result shows why enciphering with a strong PRP provides a versatile route to AE [3]. The versatility comes from the fact that the approach automatically exploits arbitrarily embedded randomness and redundancy in plaintexts for achieving privacy and authenticity. We recently came to understand just how attractive this route might be. On the one hand, we kept hearing requests for stronger AE security properties, like nonce-reuse security or security if unverified plaintexts are disclosed. Enciphering-based AE would deliver such things. On the other hand, some enciphering schemes over both long and short strings had become well-known. While they didn't have the efficiency of OCB, say, neither were they computationally exorbitant.

See the acknowledgments for some particular communications that solidified in our mind the utility of defining an enciphering-based AE scheme.

Developing the enciphering scheme. Having decided to make an enciphering-based scheme, the crucial choice was how to encipher an arbitrary-length string. With AES support increasingly embedded into devices, we wanted our method to be AES-based. Our enciphering scheme would need to be tweakable, to handle the nonce and AD, but that part would not be hard.

A wide body of work had made abundantly clear that the best techniques for AES-based enciphering depended on the length of what you were enciphering. The most practical approach for short strings was to build something that looks like a conventional cipher; for long strings, one wants something that looks more like a conventional mode. To cover all strings we'd have to glue together at least two different techniques. While this might look unusual to some, it didn't really bother us: the "EZ" in the name AEZ was supposed to suggest simplicity of correct use, not simplicity of design or even implementation.

For short strings—those under 16 bytes—some version of FFX was the obvious choice [4]. It was already in a draft standard [11] and the long history of Feistel networks made the choice seem safe (despite the fact that security bounds for balanced Feistel networks become disappointing when the input gets too short). The round function would be based on AES—or, to speed things up, a reduced-round version of it.

For longer strings, there were several off-the-shelf alternatives we could turn to. The best-known of these was EME2 [13, 15]. But its treatment of final fragments seemed complex, on top of needing two AES calls per block. Alternatives like HCTR, HEH, HMC, and TET traded one of these blockcipher calls for a potentially expensive finite-field operation, a direction we didn't want to go. We wanted to stick with AES, or its round function, for everything.

Our first plan was to encipher customary-length strings with four rounds of unbalanced Feistel (two expanding and two compressing). Both could be parallelizable, the expanding round function looking like CTR mode and the compressing round function looking like PMAC. The approach would make for an easy spec to write and implement; easy provable security claims; a clean connection to FFX; and we could even accommodate arbitrary-radix plaintexts. Yet we'd be expending four blockcipher calls per block. We felt that people would criticize the construction for its inefficiency.

To make something faster, we decided to take a fresh look at the Naor-Reingold (NR) approach [23, 24]. We could no doubt extend it to handle fractional final blocks by ciphertext stealing. (Doing this properly turned out to be nontrivial; we intentionally use a "wrong" form of ciphertext stealing [21, 31].) But the key to making AEZ fast would be to find a fast way to do the mixing. Could it be based on reduced-round AES?

The answer was *yes*, for all we really needed was to make a universal hash function, and the small MEDP (maximal expected differential probability) of four-round AES could be recast to give us just that. Namely, we could exploit the series of work that culminated in Keliher and Sui showing that $\Pr_{K,X}[\mathrm{AES4}_K(X) \oplus \mathrm{AES4}_K(X \oplus A) = B] \le 2^{-113.088}$ for any $A, B \in \{0,1\}^{128}$, $A \neq 0^{128}$ [17]. Indeed Minematsu and Tsunoo (MT) had already shown how to exploit the small MEDP of AES4 to make a parallelizable, provably-secure MAC [22]. To pass from a MAC to the needed mixing function would be easy: one just combines with an OCB-style offset sequence [18].

We backed off of using a provably-secure, parallelizable, AES4-based MAC just because it was looking like it would be complex and need a lot of independent subkeys, damaging key-agility and the size of a cryptographic context. We ended up going with a MAC that we could only heuristically justify, as explained earlier. It remains unclear just exactly how much one has to pay to get provable-security for the MAC. It's not extravagant, but what we do is smaller and simpler.

In defining AES4 we extend AES in the most standard way, omitting `MixColumns` from the final round. This was done to enable the use of APIs (like OpenSSL's) that provide no choice about the matter. In deciding which keys to use, we steal AES subkeys that an implementation would have to maintain anyway. This is a conceptually wrong thing to do—the AES4 subkeys should be independent of each other and everything else—but we were making a heuristic assumption about AMAC anyway and the likelihood of a problematic interaction seemed far-fetched.

Alternative enciphering. We considered an alternative approach for enciphering where the AES4 outputs that are xor'ed to compute AHash are also the inputs, after being offset by the AMAC output, to the ECB layer. We call this a type-2 design. Intuitively, this gets more mileage out of the mixing layer and supports reducing computational work from 18 AES rounds per block to 14 AES rounds per block Still, we eventually abandoned the idea because the specification was complex, as were assumptions necessary to prove its security. Also, implementations would have had to securely store an intermediate value as long as the plaintext itself.

No hidden weaknesses. The designers have not hidden any weaknesses in this cipher. The authors do not know any technical means by which one *could* intentionally weaken the design of a scheme like AEZ. The authors excoriate intelligence-agency efforts to subvert security standards and mass-market implementations.

6 Intellectual Property

The submitters have not applied for any patents in connection with this submission and have no intention to do so. As far as the inventors know, AEZ may be used in an application or context without IP-related restrictions. If any of this information changes, the submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

7 Consent

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

Acknowledgments

Rogaway thanks Dustin Boswell and René Struik for their unwitting role motivating the creation of AEZ. In an April 2013 email to Mihir Bellare, Boswell wrote of wanting an easier-to-use encryption scheme, and he sketched how such a scheme might look to its user. Bellare kindly passed the note on. Struik later gave a presentation that emphasized the importance of minimizing length expansion in low-energy environments [34]. Rogaway also thanks Stefan Lucks for a Jan 2012 discussion in which Lucks advocated developing a solution to the problem of leaking unverified plaintexts.

Thanks to Terence Spies feedback during the AEZ design. Work with him on FPE and FFX helped motivate linking up FPE and AE. Thanks to Tom Ristenpart and Yusi (James) Zhang for comments and corrections.

Hoang was supported by NSF grants CNS-0904380, CCF-0915675, CNS-1116800 and CNS-1228890; Krovetz was supported by NSF grant CNS-1314592; and Rogaway was supported by NSF grants CNS-1228828 and CNS-1314885. Many thanks to the NSF for their continuing support.

No animals were harmed in conducting our research.

References

[1] E. Andreeva, A. Bogdanov, A. Luykx, B. Mennink, N. Mouha, and K. Yasuda. How to securely release unverified plaintext in authenticated encryption. Cryptology ePrint Archive, Report 2014/144. Feb 25, 2014.

[2] B. Barak, Y. Dodis, H. Krawczyk, O. Pereira, K. Pietrzak, F. Standaert, and Y. Yu. Leftover Hash Lemma, Revisited *CRYPTO 2011*, LNCS vol. 6841, pp. 1–20, 2011.

[3] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. *ASIACRYPT 2000*, LNCS vol. 1976, Springer, pp. 317–330, 2000.

[4] M. Bellare, P. Rogaway, and T. Spies. The FFX mode of operation for format-preserving encryption. Draft 1.1. Submission to NIST, available from their website. Feb 20, 2010.

[5] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. *EUROCRYPT 2002*, LNCS vol. 2332, Springer, pp. 384–397, 2002.

[6] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2), pp. 143–154, 1979.

[7] L. Chen. Recommendation for key derivation through extraction-then-expansion. NIST Special Publication 800-56C. November 2011.

[8] J. Daemen and V. Rijmen. The Pelican MAC function. Cryptology ePrint Archive: Report 2005/088. 2005.

[9] J. Daemen and V. Rijmen. A New MAC construction ALRED and a Specific Instance ALPHA-MAC. *Fast Software Encryption*. LNCS vol. 3557, pp. 1–17, 2005.

[10] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC Modes. *CRYPTO 2004*, LNCS vol. 3152, Springer, pp. 494–510, 2004.

[11] M. Dworkin. Recommendation for block cipher modes of operation: methods for format-preserving encryption. NIST Special Publication 800-38G: Draft. July 2013.

[12] N. Ferguson. Authentication weaknesses in GCM. Manuscript. May 20, 2005.

[13] S. Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. INDOCRYPT 2004. pp. 315–327, 2004.

[14] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. Construction of a pseudo-random generator from any one-way function. *SIAM Journal on Computing*, 28(4), pp. 1364-1396, 1999.

[15] IEEE P1619.2. Draft standard architecture for wide-block encryption for shared storage media. 2008. Available from https://siswg.net.

[16] B. Kaliski, R. Rivest, and A. Sherman. Is DES a pure cipher? (Results of more cycling experiments on DES). *CRYPTO 85*. LNCS vol. 218, Springer, pp. 212–226, 1986.

[17] L. Keliher and J. Sui. Exact maximum expected differential and linear probability for two-round Advanced Encryption Standard. *IET Information Security*, 1(2), pp. 53–57, 2007.

[18] T. Krovetz and P. Rogaway. The software performance of authenticated-encryption modes. *FSE 2011*, LNCS vol. 6733, Springer, pp. 306–327, 2011.

[19] M. Liskov, R. Rivest, and D. Wagner. Tweakable block ciphers. *CRYPTO 2002*, LNCS vol. 2442, Springer, pp. 31–46, 2002

[20] D. McGrew. An interface and algorithms for authenticated encryption. RFC 5116. January 2008

[21] C. Meyer and S. Matyas. *Cryptography: a new dimension in computer data security*. Wiley, 1982.

[22] K. Minematsu and Y. Tsunoo. Provably secure MACs from differentially-uniform permutations and AES-based implementations. FSE 2006, LNCS vol. 4047, Springer, pp. 226–241, 2006.

[23] M. Naor and O. Reingold. On the construction of pseudo-random permutations: Luby-Rackoff revisited. *Journal of Cryptology*, 12(1), pp. 29-66, 1999.

[24] M. Naor and O. Reingold. The NR mode of operation. Undated manuscript realizing the mechanism of [23].

[25] J. Patarin. Generic attacks on Feistel schemes. *ASIACRYPT 2001*, LNCS vol. 2248, Springer, pp. 222–238, 2001. Also see Cryptology ePrint report 2008/036.

[26] J. Patarin, B. Gittins, and J. Treger. Increasing block sizes using Feistel networks: the example of the AES. *Cryptography and Security: From Theory to Applications*, LNCS vol. 6805, Springer, pp. 67–82, 2012.

[27] B. Preneel. Personal communications, via D. Bernstein. CAESAR competition: partial status of submissions (draft output of Dagstuhl discussion session 9 Jan 2014). Set of slides.

[28] P. Rogaway. Authenticated-encryption with associated-data. *ACM Conference on Computer and Communications Security*, ACM Press, pp. 98–107, 2002.

[29] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. *ASIACRYPT 2004*, LNCS vol.3329, Springer, pp. 16–31, 2004.

[30] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. *EUROCRYPT 2006*, LNCS vol. 4004, Springer, pp. 373–390, 2006. Also Cryptology ePrint Report 2006/221, retitled, Deterministic authenticated-encryption: a provable-security treatment of the key-wrap problem. 2006.

[31] P. Rogaway, M. Wooding, and H. Zhang. The security of ciphertext stealing. FSE 2012. pp. 180-195, 2012.

[32] M. Simplício, P. Barbuda, P. Barreto, T. Carvalho, and C. Margi. The MARVIN message authentication code and the LETTERSOUP authenticated encryption scheme. *Security and Communication Networks*, 2(2), pp. 165–180, 2009.

[33] M. Simplício and P. Barreto. Revisiting the security of the ALRED Design and Two of Its Variants: Marvin and LetterSoup. *IEEE Transactions on Information Theory*, 58(9), pp. 6223–6238, 2012.

[34] R. Struik. AEAD ciphers for highly constrained networks. DIAC 2013 presentation. Chicago, Illinois, USA. Aug 13, 2013.