OCB (v1)

Submitted and designed by
Ted Krovetz and **Phillip Rogaway**
ted@krovetz.net
15 March 2014

Here OCB means the algorithm recently approved for publication as an IETF RFC; the Internet-Draft that will become the RFC is in the publication queue awaiting any final edits [1]. Rather than rewrite that specification into a new format, we attach the final Internet-Draft as the bulk of this CAESAR submission. We also attach the FSE 2011 paper on which the Internet-Draft is based. It includes a security analysis, design rationale, and performance characteristics. The first few pages of this submission contain all elements not addressed by the Internet-Draft or FSE paper.

1 Specification

See the attached Internet-Draft, Sections 2–4, for a complete specification of OCB; and see Section 3 for parameter definitions and named parameter sets. There are only two parameters in OCB, defining which authenticator length and which block cipher is used.

2 Security Goals

OCB achieves two security properties, confidentiality and authenticity. Confidentiality is defined via "indistinguishability from random bits", meaning that an adversary is unable to distinguish OCB-outputs from an equal number of random bits. Authenticity is defined via "authenticity of ciphertexts", meaning that an adversary is unable to produce any valid nonce-ciphertext pair that it has not already acquired. When using OCB with a τ-bit authenticator to encrypt messages with ℓ bytes of combined plaintext and associated data, an adversary is unable to subvert privacy with probability more than $\ell^2/2^{128}$, and it is unable to subvert authenticity with probability more than $\ell^2/2^{128} + 1/2^\tau$.

Name	Key Bits	Authenticator Bits
AEAD_AES_128_OCB_TAGLEN128	128	128
AEAD_AES_128_OCB_TAGLEN96	128	96
AEAD_AES_128_OCB_TAGLEN64	128	64
AEAD_AES_192_OCB_TAGLEN128	192	128
AEAD_AES_192_OCB_TAGLEN96	192	96
AEAD_AES_192_OCB_TAGLEN64	192	64
AEAD_AES_256_OCB_TAGLEN128	256	128
AEAD_AES_256_OCB_TAGLEN96	256	96
AEAD_AES_256_OCB_TAGLEN64	256	64

The above table lists key and authenticator lengths. The various key lengths provide for different security levels when brute-force key search is a concern. The tag lengths provide different security levels against repeated forgery attempts. Integrity guarantees are provided by OCB for the nonce ("public message number"), ciphertext, and associated data. No "private message number" is employed in OCB. The cipher may lose all integrity and confidentiality if the legitimate key holder uses the same nonce to encrypt two different (plaintext, authenticated data) pairs under the same key.

3 Security Analysis

See the attached paper, "The Software Performance of Authenticated-Encryption Modes", which first appeared at FSE 2011 [2].

4 Features

OCB was designed to have the following features.

Fast. OCB is nearly as fast as CTR. Each block encryption requires just a few xor's on top of an AES call. Authentication of the ciphertext requires on average just 1.02 additional AES calls per message.

Provably secure. OCB is the result of over a decade of research and development, with several papers published in well-known outlets [2, 3, 4]. It is secure, in the sense of a nonce-based AE scheme, if its underlying blockcipher is a strong PRP.

Parallel. Most OCB computations are independent of one another, allowing both hardware and software acceleration proportional to the available computational units. For example Intel's Haswell CPU allows up to seven AES computations to be executed at once, and OCB easily exploits these resources.

Timing-attack resistant. There are no conditional computations in OCB which depend on secret data (provided the block cipher implementation also has none).

Online. One need not know all of the plaintext or associated data before beginning processing, and the two can be processed in any order. This is useful when data is streaming and when associated data is not finalized until after the plaintext is complete.

Static AD. When associated data is unchanging over a series of encryptions, the associated data's contribution need not be recalculated each time. This reduces successive computation work when multiple encryptions are associated with the same associated data.

OCB's security and feature set is similar to GCM's, but, in many settings, OCB is much more efficient in software. Also, OCB tags can be truncated to short lengths, which is not true of GCM.

5 Design Rationale

OCB is designed for minimal authentication overhead beyond what is required for provable security and simple encryption using a block cipher. It is not designed to resist nonce reuse. It is not designed to enjoy beyond-birthday-bound security. For further discussion of OCB's design rationale, see the attached FSE 2011 paper [2]. The designers have not hidden any weaknesses in this cipher and know of no hidden weaknesses.

6 Intellectual Property

Rogaway has received US patents 7,046,802, 7,200,227, 7,949,129, and 8,321,675 on OCB. These patents are freely licensed over a large space: open-source software, non-military software, and OpenSSL. See `http://www.cs.ucdavis.edu/~rogaway/ocb/license.htm` for more information.

Gligor and Donescu (VDG) and Jutla (IBM) are inventors (owners) on US patents 6,963,976, 6,973,187, 7,093,126, and 8,107,620, all which concern AE but which may or may not apply to OCB.

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 Mihir Bellare and John Black were coauthors on the CCS 2001 paper that defined the first version of OCB. Charanjit Jutla's earlier work, from Eurocrypt 2001, provided our inspiration.

References

[1] Ted Krovetz and Phillip Rogaway. The OCB Authenticated-Encryption Algorithm. IETF Internet Draft. `http://datatracker.ietf.org/doc/draft-irtf-cfrg-ocb/`. Accessed 15 March 2014.

[2] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. Fast Software Encryption 2011 (FSE 2011). Lecture Notes in Computer Science, Springer, 2011.

[3] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. Advances in Cryptology — ASIACRYPT 2004. Lecture Notes in Computer Science, vol. 3329, Springer, pp. 16—31, 2004.

[4] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. ACM Conference on Computer and Communications Security (CCS 2001), ACM Press, pp. 196–205, 2001.

Internet Research Task Force
Internet-Draft
Intended status: Informational
Expires: August 10, 2014

T. Krovetz
Sacramento State
P. Rogaway
UC Davis

February 06, 2014

The OCB Authenticated-Encryption Algorithm
draft-irtf-cfrg-ocb-07

Abstract

This document specifies OCB, a shared-key blockcipher-based encryption scheme that provides confidentiality and authenticity for plaintexts and authenticity for associated data. This document is a product of the Crypto Forum Research Group (CFRG).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Schemes for authenticated encryption (AE) simultaneously provide for confidentiality and authentication. While this goal would traditionally be achieved by melding separate encryption and authentication mechanisms, each using its own key, integrated AE schemes intertwine what is needed for confidentiality and what is needed for authenticity. By conceptualizing AE as a single cryptographic goal, AE schemes are less likely to be misused than conventional encryption schemes. Also, integrated AE schemes can be significantly faster than what one sees from composing separate confidentiality and authenticity means.

When an AE scheme allows for the authentication of unencrypted data at the same time that a plaintext is being encrypted and authenticated, the scheme is an authenticated encryption with associated data (AEAD) scheme. Associated data can be useful when, for example, a network packet has unencrypted routing information and an encrypted payload.

OCB (short for Offset Codebook) is an AEAD scheme that depends on a blockcipher. This document fully defines OCB encryption and decryption except for the choice of the blockcipher and the length of authentication tag that is part of the ciphertext. The blockcipher must have a 128-bit blocksize. Each choice of blockcipher and tag length specifies a different variant of OCB. Several AES-based variants are defined in Section 3.1.

OCB encryption and decryption employ a nonce N, which must be distinct for each invocation of the OCB encryption operation. OCB requires the associated data A to be specified when one encrypts or decrypts, but it may be zero-length. The plaintext P and the associated data A can have any bitlength. The ciphertext C one gets by encrypting P in the presence of A consists of a ciphertext-core having the same length as P, plus an authentication tag. One can view the resulting ciphertext as either the pair (ciphertext-core, tag) or their concatenation (ciphertext-core || tag), the difference being purely how one assembles and parses ciphertexts. This document uses concatenation.

OCB encryption protects the confidentiality of P and the authenticity of A, N, and P. It does this using, on average, about a + m + 1.02 blockcipher calls, where a is the blocklength of A and m is the blocklength of P and the nonce N is implemented as a counter (if N is random then OCB uses a + m + 2 blockcipher calls). If A is fixed during a session then, after preprocessing, there is effectively no cost to having A authenticated on subsequent encryptions, and the mode will average m + 1.02 blockcipher calls. OCB requires a single key K for the underlying blockcipher, and all blockcipher calls are keyed by K. OCB is online. In particular, one need not know the length of A or P to proceed with encryption, nor need one know the length of A or C to proceed with decryption. OCB is parallelizable: the bulk of its blockcipher calls can be performed simultaneously. Computational work beyond blockcipher calls consists of a small and fixed number of logical operations per call. OCB enjoys provable security: the mode of operation is secure assuming that the underlying blockcipher is secure. As with most modes of operation, security degrades as the number of blocks processed gets large (see Section 5 for details).

For reasons of generality, OCB is defined to operate on arbitrary bit-strings. But for reasons of simplicity and efficiency, most implementations will assume that strings operated on are byte-strings (ie, strings that are a multiple of 8 bits). To promote interoperability, implementations of OCB that communicate with implementations of unknown capabilities should restrict all provided values (nonces, tags, plaintexts, ciphertexts, and associated data) to byte-strings.

The version of OCB defined in this document is a refinement of two prior schemes. The original OCB version was published in 2001 [OCB1] and was listed as an optional component in IEEE 802.11i. A second version was published in 2004 [OCB2] and is specified in ISO 19772. The scheme described here is called OCB3 in the 2011 paper describing the mode [OCB3]; it shall be referred to simply as OCB throughout this document. The only difference between the algorithm of this RFC and that of the [OCB3] paper is that the tag length is now encoded into the internally formatted nonce. See [OCB3] for complete references, timing information, and a discussion of the differences between the algorithms.

OCB has received years of in-depth analysis previous to its submission to the CFRG, and has been under review by the members of the CFRG for over a year. It is the consensus of the CFRG that the security mechanisms provided by the OCB AEAD algorithm described in this document are suitable for use in providing confidentiality and authentication.

2. Notation and Basic Operations

There are two types of variables used in this specification, strings and integers. Although strings processed by most implementations of OCB will be strings of bytes, bit-level operations are used throughout this specification document for defining OCB. String variables are always written with an initial upper-case letter while integer variables are written in all lower-case. Following C's convention, a single equals ("=") indicates variable assignment and double equals ("==") is the equality relation. Whenever a variable is followed by an underscore ("_"), the underscore is intended to denote a subscript, with the subscripted expression requiring evaluation to resolve the meaning of the variable. For example, when i == 2, then P_i refers to the variable P_2.

c^i — The integer c raised to the i-th power.

bitlen(S) — The length of string S in bits (eg, bitlen(101) == 3).

zeros(n) — The string made of n zero-bits.

ntz(n) — The number of trailing zero bits in the base-2 representation of the positive integer n. More formally, ntz(n) is the largest integer x for which 2^x divides n.

S xor T — The string that is the bitwise exclusive-or of S and T. Strings S and T will always have the same length.

S[i] — The i-th bit of the string S (indices begin at 1, so if S is 011 then S[1] == 0, S[2] == 1, S[3] == 1).

S[i..j] — The substring of S consisting of bits i through j, inclusive.

S || T — String S concatenated with string T (eg, 000 || 111 == 000111).

str2num(S) — The base-2 interpretation of bitstring S (eg, str2num(1110) == 14).

num2str(i,n) — The n-bit string whose base-2 interpretation is i (eg, num2str(14,4) == 1110 and num2str(1,2) == 01).

double(S) — If S[1] == 0 then double(S) == (S[2..128] || 0); otherwise double(S) == (S[2..128] || 0) xor (zeros(120) || 10000111).

3. OCB Global Parameters

To be complete, the algorithms in this document require specification of two global parameters: a blockcipher operating on 128-bit blocks and the length of authentication tags in use.

Specifying a blockcipher implicitly defines the following symbols.

KEYLEN — The blockcipher's key length, in bits.

ENCIPHER(K,P) — The blockcipher function mapping 128-bit plaintext block P to its corresponding ciphertext block using KEYLEN-bit key K.

DECIPHER(K,C) — The inverse blockcipher function mapping 128-bit ciphertext block C to its corresponding plaintext block using KEYLEN-bit key K.

The TAGLEN parameter specifies the length of authentication tag used by OCB and may be any value up to 128. Greater values for TAGLEN provide greater assurances of authenticity, but ciphertexts produced by OCB are longer than their corresponding plaintext by TAGLEN bits. See Section 5 for details about TAGLEN and security.

As an example, if 128-bit authentication tags and AES with 192-bit keys are to be used, then KEYLEN is 192, ENCIPHER refers to the AES-192 cipher, DECIPHER refers to the AES-192 inverse cipher, and TAGLEN is 128 [AES].

3.1. Named OCB Parameter Sets and RFC 5116 Constants

The following table gives names to common OCB global parameter sets. Each of the AES variants is defined in [AES].

Name	Blockcipher	TAGLEN
AEAD_AES_128_OCB_TAGLEN128	AES-128	128
AEAD_AES_128_OCB_TAGLEN96	AES-128	96
AEAD_AES_128_OCB_TAGLEN64	AES-128	64
AEAD_AES_192_OCB_TAGLEN128	AES-192	128
AEAD_AES_192_OCB_TAGLEN96	AES-192	96
AEAD_AES_192_OCB_TAGLEN64	AES-192	64
AEAD_AES_256_OCB_TAGLEN128	AES-256	128
AEAD_AES_256_OCB_TAGLEN96	AES-256	96
AEAD_AES_256_OCB_TAGLEN64	AES-256	64

RFC 5116 defines an interface for authenticated encryption schemes [RFC5116]. RFC 5116 requires the specification of certain constants for each named AEAD scheme. For each of the OCB parameter sets listed above: P_MAX, A_MAX, and C_MAX are all unbounded; N_MIN is 1 byte and N_MAX is 15 bytes. The parameter-sets indicating the use of AES-128, AES-192 and AES-256 have K_LEN equal to 16, 24 and 32 bytes, respectively.

Each ciphertext is longer than its corresponding plaintext by exactly TAGLEN bits, and TAGLEN is given at the end of each name. For instance, an AEAD_AES_128_OCB_TAGLEN64 ciphertext is exactly 64 bits longer than its corresponding plaintext.

4. OCB Algorithms

OCB is described in this section using pseudocode. Given any collection of inputs of the required types, following the pseudocode description for a function will produce the correct output of the promised type.

4.1. Associated-Data Processing: HASH

OCB has the ability to authenticate unencrypted associated data at the same time that it provides for authentication and encrypts a plaintext. The following hash function is central to providing this functionality. If an application has no associated data, then the associated data should be considered to exist and to be the empty string. HASH, conveniently, always returns zeros(128) when the associated data is the empty string.

```
Function name:
  HASH
Input:
```

```
     K, string of KEYLEN bits                      // Key
     A, string of any length                       // Associated data
   Output:
     Sum, string of 128 bits                       // Hash result
```

Sum is defined as follows.

```
     //
     // Key-dependent variables
     //
     L_* = ENCIPHER(K, zeros(128))
     L_$ = double(L_*)
     L_0 = double(L_$)
     L_i = double(L_{i-1}) for every integer i > 0

     //
     // Consider A as a sequence of 128-bit blocks
     //
     Let m be the largest integer so that 128m <= bitlen(A)
     Let A_1, A_2, ..., A_m and A_* be strings so that
       A == A_1 || A_2 || ... || A_m || A_*, and
       bitlen(A_i) == 128 for each 1 <= i <= m.
       Note: A_* may possibly be the empty string.

     //
     // Process any whole blocks
     //
     Sum_0 = zeros(128)
     Offset_0 = zeros(128)
     for each 1 <= i <= m
        Offset_i = Offset_{i-1} xor L_{ntz(i)}
        Sum_i = Sum_{i-1} xor ENCIPHER(K, A_i xor Offset_i)
     end for

     //
     // Process any final partial block; compute final hash value
     //
     if bitlen(A_*) > 0 then
        Offset_* = Offset_m xor L_*
        CipherInput = (A_* || 1 || zeros(127-bitlen(A_*))) xor Offset_*
        Sum = Sum_m xor ENCIPHER(K, CipherInput)
     else
        Sum = Sum_m
     end if
```

4.2. Encryption: OCB-ENCRYPT

This function computes a ciphertext (which includes a bundled authentication tag) when given a plaintext, associated data, nonce and key. For each invocation of OCB-ENCRYPT using the same key K, the value of the nonce input N must be distinct.

```
Function name:
  OCB-ENCRYPT
Input:
  K, string of KEYLEN bits                      // Key
  N, string of no more than 120 bits            // Nonce
  A, string of any length                       // Associated data
  P, string of any length                       // Plaintext
Output:
  C, string of length bitlen(P) + TAGLEN bits   // Ciphertext
```

C is defined as follows.

```
  //
  // Key-dependent variables
  //
  L_* = ENCIPHER(K, zeros(128))
  L_$ = double(L_*)
  L_0 = double(L_$)
  L_i = double(L_{i-1}) for every integer i > 0

  //
  // Consider P as a sequence of 128-bit blocks
  //
  Let m be the largest integer so that 128m <= bitlen(P)
  Let P_1, P_2, ..., P_m and P_* be strings so that
    P == P_1 || P_2 || ... || P_m || P_*, and
    bitlen(P_i) == 128 for each 1 <= i <= m.
    Note: P_* may possibly be the empty string.

  //
  // Nonce-dependent and per-encryption variables
  //
  Nonce = num2str(TAGLEN mod 128,7) || zeros(120-bitlen(N)) || 1 || N
  bottom = str2num(Nonce[123..128])
  Ktop = ENCIPHER(K, Nonce[1..122] || zeros(6))
  Stretch = Ktop || (Ktop[1..64] xor Ktop[9..72])
  Offset_0 = Stretch[1+bottom..128+bottom]
  Checksum_0 = zeros(128)

  //
  // Process any whole blocks
  //
  for each 1 <= i <= m
```

```
      Offset_i = Offset_{i-1} xor L_{ntz(i)}
      C_i = Offset_i xor ENCIPHER(K, P_i xor Offset_i)
      Checksum_i = Checksum_{i-1} xor P_i
   end for

   //
   // Process any final partial block and compute raw tag
   //
   if bitlen(P_*) > 0 then
      Offset_* = Offset_m xor L_*
      Pad = ENCIPHER(K, Offset_*)
      C_* = P_* xor Pad[1..bitlen(P_*)]
      Checksum_* = Checksum_m xor (P_* || 1 || zeros(127-bitlen(P_*)))
      Tag = ENCIPHER(K, Checksum_* xor Offset_* xor L_$) xor HASH(K,A)
   else
      C_* = <empty string>
      Tag = ENCIPHER(K, Checksum_m xor Offset_m xor L_$) xor HASH(K,A)
   end if

   //
   // Assemble ciphertext
   //
   C = C_1 || C_2 || ... || C_m || C_* || Tag[1..TAGLEN]
```

4.3. Decryption: OCB-DECRYPT

This function computes a plaintext when given a ciphertext, associated data, nonce and key. An authentication tag is embedded in the ciphertext. If the tag is not correct for the ciphertext, associated data, nonce and key, then an INVALID signal is produced.

```
Function name:
  OCB-DECRYPT
Input:
  K, string of KEYLEN bits                      // Key
  N, string of no more than 120 bits            // Nonce
  A, string of any length                       // Associated data
  C, string of at least TAGLEN bits             // Ciphertext
Output:
  P, string of length bitlen(C) - TAGLEN bits,  // Plaintext
       or INVALID indicating authentication failure
```

P is defined as follows.

```
   //
   // Key-dependent variables
   //
```

```
L_* = ENCIPHER(K, zeros(128))
L_$ = double(L_*)
L_0 = double(L_$)
L_i = double(L_{i-1}) for every integer i > 0

//
// Consider C as a sequence of 128-bit blocks
//
Let m be the largest integer so that 128m <= bitlen(C) - TAGLEN
Let C_1, C_2, ..., C_m, C_* and T be strings so that
  C == C_1 || C_2 || ... || C_m || C_* || T,
  bitlen(C_i) == 128 for each 1 <= i <= m, and
  bitlen(T) == TAGLEN.
  Note: C_* may possibly be the empty string.

//
// Nonce-dependent and per-decryption variables
//
Nonce = num2str(TAGLEN mod 128,7) || zeros(120-bitlen(N)) || 1 || N
bottom = str2num(Nonce[123..128])
Ktop = ENCIPHER(K, Nonce[1..122] || zeros(6))
Stretch = Ktop || (Ktop[1..64] xor Ktop[9..72])
Offset_0 = Stretch[1+bottom..128+bottom]
Checksum_0 = zeros(128)

//
// Process any whole blocks
//
for each 1 <= i <= m
   Offset_i = Offset_{i-1} xor L_{ntz(i)}
   P_i = Offset_i xor DECIPHER(K, C_i xor Offset_i)
   Checksum_i = Checksum_{i-1} xor P_i
end for

//
// Process any final partial block and compute raw tag
//
if bitlen(C_*) > 0 then
   Offset_* = Offset_m xor L_*
   Pad = ENCIPHER(K, Offset_*)
   P_* = C_* xor Pad[1..bitlen(C_*)]
   Checksum_* = Checksum_m xor (P_* || 1 || zeros(127-bitlen(P_*)))
   Tag = ENCIPHER(K, Checksum_* xor Offset_* xor L_$) xor HASH(K,A)
else
   P_* = <empty string>
   Tag = ENCIPHER(K, Checksum_m xor Offset_m xor L_$) xor HASH(K,A)
end if
```

```
//
// Check for validity and assemble plaintext
//
if (Tag[1..TAGLEN] == T) then
   P = P_1 || P_2 || ... || P_m || P_*
else
   P = INVALID
end if
```

5. Security Considerations

OCB achieves two security properties, confidentiality and authenticity. Confidentiality is defined via "indistinguishability from random bits", meaning that an adversary is unable to distinguish OCB-outputs from an equal number of random bits. Authenticity is defined via "authenticity of ciphertexts", meaning that an adversary is unable to produce any valid nonce-ciphertext pair that it has not already acquired. The security guarantees depend on the underlying blockcipher being secure in the sense of a strong pseudorandom permutation. Thus if OCB is used with a blockcipher that is not secure as a strong pseudorandom permutation, the security guarantees vanish. The need for the strong pseudorandom permutation property means that OCB should be used with a conservatively designed, well-trusted blockcipher, such as AES.

Both the confidentiality and the authenticity properties of OCB degrade as per s^2 / 2^128, where s is the total number of blocks that the adversary acquires. The consequence of this formula is that the proven security disappears when s becomes as large as 2^64. Thus the user should never use a key to generate an amount of ciphertext that is near to, or exceeds, 2^64 blocks. In order to ensure that s^2 / 2^128 remains small, a given key should be used to encrypt at most 2^48 blocks (2^55 bits or 4 petabytes), including the associated data. To ensure these limits are not crossed, automated key management is recommended in systems exchanging large amounts of data [RFC4107].

When a ciphertext decrypts as INVALID it is the implementor's responsibility to make sure that no information beyond this fact is made adversarially available.

OCB encryption and decryption produce an internal 128-bit authentication tag. The parameter TAGLEN determines how many bits of this internal tag are included in ciphertexts and used for authentication. The value of TAGLEN has two impacts: An adversary can trivially forge with probability 2^{-TAGLEN}, and ciphertexts are TAGLEN bits longer than their corresponding plaintexts. It is up to the application designer to choose an appropriate value for TAGLEN. Long tags cost no more computationally than short ones.

Normally, a given key should be used to create ciphertexts with a single tag length, TAGLEN, and an application should reject any ciphertext that claims authenticity under the same key but a different tag length. While the ciphertext core and all of the bits of the tag do depend on the tag length, this is done for added robustness to misuse and should not suggest that receivers accept ciphertexts employing variable tag lengths under a single key.

Timing attacks are not a part of the formal security model and an implementation should take care to mitigate them in contexts where this is a concern. To render timing attacks impotent, the amount of time to encrypt or decrypt a string should be independent of the key and the contents of the string. The only explicitly conditional OCB operation that depends on private data is double(), which means that using constant-time blockcipher and double() implementations eliminates most (if not all) sources of timing attacks on OCB. Power-usage attacks are likewise out of scope of the formal model, and should be considered for environments where they are threatening.

The OCB encryption scheme reveals in the ciphertext the length of the plaintext. Sometimes the length of the plaintext is a valuable piece of information that should be hidden. For environments where "traffic analysis" is a concern, techniques beyond OCB encryption (typically involving padding) would be necessary.

Defining the ciphertext that results from OCB-ENCRYPT to be the pair (C_1 || C_2 || ... || C_m || C_*, Tag[1..TAGLEN]) instead of the concatenation C_1 || C_2 || ... || C_m || C_* || Tag[1..TAGLEN] introduces no security concerns. Because TAGLEN is fixed, both versions allow ciphertexts to be parsed unambiguously.

5.1. Nonce Requirements

It is crucial that, as one encrypts, one does not repeat a nonce. The inadvertent reuse of the same nonce by two invocations of the OCB encryption operation, with the same key, but with distinct plaintext values, undermines the confidentiality of the plaintexts protected in those two invocations, and undermines all of the authenticity and integrity protection provided by that key. For this reason, OCB should only be used whenever nonce uniqueness can be provided with certainty. Note that it is acceptable to input the same nonce value multiple times to the decryption operation. We emphasize that the security consequences are quite serious if an attacker observes two ciphertexts that were created using the same nonce and key values, unless the plaintext and AD values in both invocations of the encrypt operation were identical. First, a loss of confidentiality ensues because the attacker will be able to infer relationships between the two plaintext values. Second, a loss of authenticity ensues because the attacker will be able to recover secret information used to provide authenticity, making subsequent forgeries trivial. Note that there are AEAD schemes, particularly SIV [RFC5297], appropriate for environments where nonces are unavailable or unreliable. OCB is not such a scheme.

Nonces need not be secret, and a counter may be used for them. If two parties send OCB-encrypted plaintexts to one another using the same key, then the space of nonces used by the two parties must be partitioned so that no nonce that could be used by one party to encrypt could be used by the other to encrypt (eg, odd and even counters).

6. IANA Considerations

The Internet Assigned Numbers Authority (IANA) has defined a registry for Authenticated Encryption with Associated Data parameters. The IANA has added the following entries to the AEAD Registry. Each name refers to a set of parameters defined in Section 3.1.

Name	Reference	Numeric Identifier
AEAD_AES_128_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_128_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_128_OCB_TAGLEN64	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN64	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN64	Section 3.1	XX

7. Acknowledgements

The design of the original OCB scheme [OCB1] was done while Rogaway was at Chiang Mai University, Thailand. Follow-up work [OCB2] was done with support of NSF grant 0208842 and a gift from Cisco. The final work by Krovetz and Rogaway [OCB3] that has resulted in this spec was supported by NSF grant 0904380. Thanks go to the many members of the Crypto Forum Research Group (CFRG) who provided feedback on earlier drafts. Thanks in particular go to David McGrew for contributing some text and for managing the RFC approval process, to James Manger for initiating a productive discussion on tag-length dependency and for greatly improving Appendix A, to Matt Caswell and Peter Dettman for writing implementations and verifying test vectors, and to Stephen Farrell and Spencer Dawkins for their careful reading and suggestions.

8. References

8.1. Normative References

[RFC5116] McGrew, D., "An interface and algorithms for authenticated encryption", RFC 5116, January 2008.

[AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

8.2. Informative References

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for cryptographic key management", RFC 4107, June 2005.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) authenticated encryption using the Advanced Encryption Standard (AES)", RFC 5297, October 2008.

[OCB3] Krovetz, T. and P. Rogaway, "The software performance of authenticated-encryption modes", in Fast Software Encryption - FSE 2011, Springer, 2011.

[OCB2] Rogaway, P., "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC", in Advances in Cryptology - ASIACRYPT 2004, Springer, 2004.

[OCB1] Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB: a block-cipher mode of operation for efficient authenticated encryption", in ACM Conference on Computer and Communications Security 2001 - CCS 2001, ACM Press, 2001.

[OCB1ALT] Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB Mode: proposal to NIST for a block-cipher mode of operation which simultaneously provides privacy and authenticity", at http://eprint.iacr.org/2001/026, IACR ePrint Archive, 2001.

Appendix A. Sample Results

This section gives sample output values for various inputs when using OCB with AES as per the parameters defined in Section 3.1. All strings are represented in hexadecimal (eg, 0F represents the bitstring 00001111).

The following 16 (N,A,P,C) tuples show the ciphertext C that results from OCB-ENCRYPT(K,N,A,P) for various lengths of associated data (A) and plaintext (P). The key (K) has a fixed value, the tag length is 128 bits, and the the nonce (N) increments.

```
K : 000102030405060708090A0B0C0D0E0F
```

An empty entry indicates the empty string.

```
N: BBAA99887766554433221100
A:
P:
C: 785407BFFFC8AD9EDCC5520AC9111EE6

N: BBAA99887766554433221101
A: 0001020304050607
P: 0001020304050607
C: 6820B3657B6F615A5725BDA0D3B4EB3A257C9AF1F8F03009

N: BBAA99887766554433221102
A: 0001020304050607
P:
C: 81017F8203F081277152FADE694A0A00

N: BBAA99887766554433221103
A:
P: 0001020304050607
C: 45DD69F8F5AAE72414054CD1F35D82760B2CD00D2F99BFA9

N: BBAA99887766554433221104
A: 000102030405060708090A0B0C0D0E0F
P: 000102030405060708090A0B0C0D0E0F
C: 571D535B60B277188BE5147170A9A22C3AD7A4FF3835B8C5
   701C1CCEC8FC3358

N: BBAA99887766554433221105
A: 000102030405060708090A0B0C0D0E0F
P:
C: 8CF761B6902EF764462AD86498CA6B97

N: BBAA99887766554433221106
A:
```

```
      P: 000102030405060708090A0B0C0D0E0F
      C: 5CE88EC2E0692706A915C00AEB8B2396F40E1C743F52436B
         DF06D8FA1ECA343D

      N: BBAA99887766554433221107
      A: 000102030405060708090A0B0C0D0E0F1011121314151617
      P: 000102030405060708090A0B0C0D0E0F1011121314151617
      C: 1CA2207308C87C010756104D8840CE1952F09673A448A122
         C92C62241051F57356D7F3C90BB0E07F

      N: BBAA99887766554433221108
      A: 000102030405060708090A0B0C0D0E0F1011121314151617
      P:
      C: 6DC225A071FC1B9F7C69F93B0F1E10DE

      N: BBAA99887766554433221109
      A:
      P: 000102030405060708090A0B0C0D0E0F1011121314151617
      C: 221BD0DE7FA6FE993ECCD769460A0AF2D6CDED0C395B1C3C
         E725F32494B9F914D85C0B1EB38357FF

      N: BBAA9988776655443322110A
      A: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F
      P: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F
      C: BD6F6C496201C69296C11EFD138A467ABD3C707924B964DE
         AFFC40319AF5A48540FBBA186C5553C68AD9F592A79A4240

      N: BBAA9988776655443322110B
      A: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F
      P:
      C: FE80690BEE8A485D11F32965BC9D2A32

      N: BBAA9988776655443322110C
      A:
      P: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F
      C: 2942BFC773BDA23CABC6ACFD9BFD5835BD300F0973792EF4
         6040C53F1432BCDFB5E1DDE3BC18A5F840B52E653444D5DF

      N: BBAA9988776655443322110D
      A: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F2021222324252627
      P: 000102030405060708090A0B0C0D0E0F1011121314151617
         18191A1B1C1D1E1F2021222324252627
      C: D5CA91748410C1751FF8A2F618255B68A0A12E093FF45460
```

```
      6E59F9C1D0DDC54B65E8628E568BAD7AED07BA06A4A69483
      A7035490C5769E60

   N: BBAA9988776655443322110E
   A: 000102030405060708090A0B0C0D0E0F1011121314151617
      18191A1B1C1D1E1F2021222324252627
   P:
   C: C5CD9D1850C141E358649994EE701B68

   N: BBAA9988776655443322110F
   A:
   P: 000102030405060708090A0B0C0D0E0F1011121314151617
      18191A1B1C1D1E1F2021222324252627
   C: 4412923493C57D5DE0D700F753CCE0D1D2D95060122E9F15
      A5DDBFC5787E50B5CC55EE507BCB084E479AD363AC366B95
      A98CA5F3000B1479
```

Next are several internal values generated during the OCB-ENCRYPT computation for the last test vector listed above.

```
L_*       : C6A13B37878F5B826F4F8162A1C8D879
L_$       : 8D42766F0F1EB704DE9F02C54391B075
L_0       : 1A84ECDE1E3D6E09BD3E058A8723606D
L_1       : 3509D9BC3C7ADC137A7C0B150E46C0DA
bottom    : 15 (decimal)
Ktop      : 9862B0FDEE4E2DD56DBA6433F0125AA2
Stretch   : 9862B0FDEE4E2DD56DBA6433F0125AA2FAD24D13A063F8B8
Offset_0  : 587EF72716EAB6DD3219F8092D517D69
Offset_1  : 42FA1BF908D7D8D48F27FD83AA721D04
Offset_2  : 77F3C24534AD04C7F55BF696A434DDDE
Offset_*  : B152F972B3225F459A1477F405FC05A7
Checksum_1: 000102030405060708090A0B0C0D0E0F
Checksum_2: 10101010101010101010101010101010
Checksum_*: 30313233343536379010101010101010
```

The next tuple shows a result with a tag length of 96 bits, and a different key.

```
   K: 0F0E0D0C0B0A09080706050403020100

   N: BBAA9988776655443322110D
   A: 000102030405060708090A0B0C0D0E0F1011121314151617
      18191A1B1C1D1E1F2021222324252627
   P: 000102030405060708090A0B0C0D0E0F1011121314151617
      18191A1B1C1D1E1F2021222324252627
   C: 1792A4E31E0755FB03E31B22116E6C2DDF9EFD6E33D536F1
```

```
A0124B0A55BAE884ED93481529C76B6AD0C515F4D1CDD4FD
AC4F02AA
```

The following algorithm tests a wider variety of inputs. Results are given for each parameter set defined in Section 3.1.

```
K = zeros(KEYLEN-8) || num2str(TAGLEN,8)
C = <empty string>
for i = 0 to 127 do
   S = zeros(8i)
   N = num2str(3i+1,96)
   C = C || OCB-ENCRYPT(K,N,S,S)
   N = num2str(3i+2,96)
   C = C || OCB-ENCRYPT(K,N,<empty string>,S)
   N = num2str(3i+3,96)
   C = C || OCB-ENCRYPT(K,N,S,<empty string>)
end for
N = num2str(385,96)
Output : OCB-ENCRYPT(K,N,C,<empty string>)
```

Iteration i of the loop adds 2i + (3 * TAGLEN / 8) bytes to C, resulting in an ultimate length for C of 22,400 bytes when TAGLEN == 128, 20,864 bytes when TAGLEN == 192, and 19,328 bytes when TAGLEN == 64. The final OCB-ENCRYPT has an empty plaintext component, so serves only to authenticate C. The output should be:

```
AEAD_AES_128_OCB_TAGLEN128 Output: 67E944D23256C5E0B6C61FA22FDF1EA2
AEAD_AES_192_OCB_TAGLEN128 Output: F673F2C3E7174AAE7BAE986CA9F29E17
AEAD_AES_256_OCB_TAGLEN128 Output: D90EB8E9C977C88B79DD793D7FFA161C
AEAD_AES_128_OCB_TAGLEN96 Output : 77A3D8E73589158D25D01209
AEAD_AES_192_OCB_TAGLEN96 Output : 05D56EAD2752C86BE6932C5E
AEAD_AES_256_OCB_TAGLEN96 Output : 5458359AC23B0CBA9E6330DD
AEAD_AES_128_OCB_TAGLEN64 Output : 192C9B7BD90BA06A
AEAD_AES_192_OCB_TAGLEN64 Output : 0066BC6E0EF34E24
AEAD_AES_256_OCB_TAGLEN64 Output : 7D4EA5D445501CBE
```

Authors' Addresses

Ted Krovetz
Computer Science Department
California State University
6000 J Street
Sacramento, CA 95819-6021
USA

Email: ted@krovetz.net

Phillip Rogaway
Computer Science Department
University of California
One Shields Avenue
Davis, CA 95616-8562
USA

Email: rogaway@cs.ucdavis.edu

An earlier version of this paper appears at *Fast Software Encryption 2011* (FSE 2011).

The Software Performance of Authenticated-Encryption Modes

Ted Krovetz* Phillip Rogaway†

March 21, 2011

Abstract

We study the software performance of authenticated-encryption modes CCM, GCM, and OCB. Across a variety of platforms, we find OCB to be substantially faster than either alternative. For example, on an Intel i5 ("Clarkdale") processor, good implementations of CCM, GCM, and OCB encrypt at around 4.2 cpb, 3.7 cpb, and 1.5 cpb, while CTR mode requires about 1.3 cpb. Still we find room for algorithmic improvements to OCB, showing how to trim one blockcipher call (most of the time, assuming a counter-based nonce) and reduce latency. Our findings contrast with those of McGrew and Viega (2004), who claimed similar performance for GCM and OCB.

Key words: authenticated encryption, cryptographic standards, encryption speed, modes of operation, CCM, GCM, OCB.

1 Introduction

Background. Over the past few years, considerable effort has been spent constructing schemes for *authenticated encryption* (AE). One reason is recognition of the fact that a scheme that delivers both privacy and authenticity may be more efficient than the straightforward amalgamation of separate privacy and authenticity techniques. A second reason is the realization that an AE scheme is less likely to be incorrectly used than an encryption scheme designed for privacy alone.

While other possibilities exist, it is natural to build AE schemes from blockciphers, employing some *mode of operation*. There are two approaches. In a *composed* ("two-pass") AE scheme one conjoins essentially separate privacy and authenticity modes. For example, one might apply CTR-mode encryption and then compute some version of the CBC MAC. Alternatively, in an *integrated* ("one-pass") AE scheme the parts of the mechanism responsible for privacy and for authenticity are tightly coupled.[1] Such schemes emerged around a decade ago, with the work of Jutla [21], Katz and Yung [23], and Gligor and Donescu [11].

Integrated AE schemes were invented to improve performance of composed ones, but it has not been clear if they do. In the only comparative study to date [31], McGrew and Viega found that their composed scheme, GCM, was about as fast as, and sometimes faster than, the integrated scheme OCB [35] (hereinafter OCB1, to distinguish it from a subsequent variant we'll call OCB2 [34]).

* Computer Science Department, California State University, 6000 J Street, Sacramento, California 95819, USA. E-mail: tdk@acm.org, URL: `http://krovetz.net/csus`

† Department of Computer Science, Kemper Hall of Engineering, University of California, Davis, California 95616, USA. E-mail: rogaway@cs.ucdavis.edu, URL: `http://www.cs.ucdavis.edu/~rogaway`

[1] The distinction between composed and integrated AE schemes is useful but not formal.

scheme	ref	date	ty	high-level description	standard
EtM	[1]	2000	C	Encrypt-then-MAC (and other) generic comp. schemes	ISO 19772
RPC	[23]	2000	I	Insert counters and sentinels in blocks, then ECB	—
IAPM	[21]	2001	I	Seminal integrated scheme. Also IACBC	—
XCBC	[11]	2001	I	Concurrent with Jutla's work. Also XECB	—
✓ OCB1	[35]	2001	I	Optimized design similar to IAPM	—
TAE	[28]	2002	I	Recasts OCB1 using a tweakable blockcipher	—
✓ CCM	[39]	2002	C	CTR encryption + CBC MAC	NIST 800-38C
CWC	[24]	2004	C	CTR encryption + GF($2^{127}-1$)-based CW MAC	—
✓ GCM	[31]	2004	C	CTR encryption + GF(2^{128})-based CW MAC	NIST 800-38D
EAX	[2]	2004	C	CTR encryption + CMAC, a cleaned-up CCM	ISO 19772
✓ OCB2	[34]	2004	I	OCB1 with AD and alleged speed improvements	ISO 19772
CCFB	[29]	2005	I	Similar to RPC [23], but with chaining	—
CHM	[18]	2006	C	Beyond-birthday-bound security	—
SIV	[36]	2006	C	Deterministic/misuse-resistant AE	RFC 5297
CIP	[17]	2008	C	Beyond-birthday-bound security	—
HBS	[20]	2009	C	Deterministic AE. Single key	—
BTM	[19]	2009	C	Deterministic AE. Single key, no blockcipher inverse	—
✓ OCB3	new	2010	I	Refines the prior versions of OCB	—

Figure 1: **Authenticated-encryption schemes built from a blockcipher.** Checks ✓ indicate schemes included in our performance study. The column labeled **ty** (type) specifies if the scheme is integrated (I) or composed (C). When a scheme is in multiple standards, only a single one is named.

After McGrew and Viega's 2004 paper, no subsequent performance study was ever published. This is unfortunate, as there seems to have been a major problem with their work: reference implementations were compared against optimized ones, and none of the results are repeatable due to the use of proprietary code. In the meantime, CCM and GCM have become quite important to cryptographic practice. For example, CCM underlies modern WiFi (802.11i) security, while GCM is supported in IPsec and TLS.

McGrew and Viega identified two performance issues in the design of OCB1. First, the mode uses $m+2$ blockcipher calls to encrypt a message of $m = \lceil |M|/128 \rceil$ blocks. In contrast, GCM makes do with $m+1$ blockcipher calls. Second, OCB1 twice needs one AES result before another AES computation can proceed. Both in hardware and in software, this can degrade performance. Beyond these facts, existing integrated modes cannot exploit the "locality" of counters in CTR mode—that high-order bits of successive input blocks are usually unchanged, an observation first exploited, for software speed, by Hongjun Wu [4]. Given all of these concerns, maybe GCM really is faster than OCB—and, more generally, maybe composed schemes are the fastest way to go. The existence of extremely high-speed MACs supports this possibility [3, 5, 25].

Contributions. We begin by refining the definition of OCB to address the performance concerns just described. When the provided nonce is a counter, the mode that we call OCB3 shaves off one AES encipherment per message encrypted about 98% of the time. In saying that the nonce is a counter we mean that, in a given session, its top portion stays fixed, while, with each successive message, the bottom portion gets bumped by one. This is the approach recommended in RFC 5116 [30, Section 3.2] and, we believe, the customary way to use an AE scheme. We do not

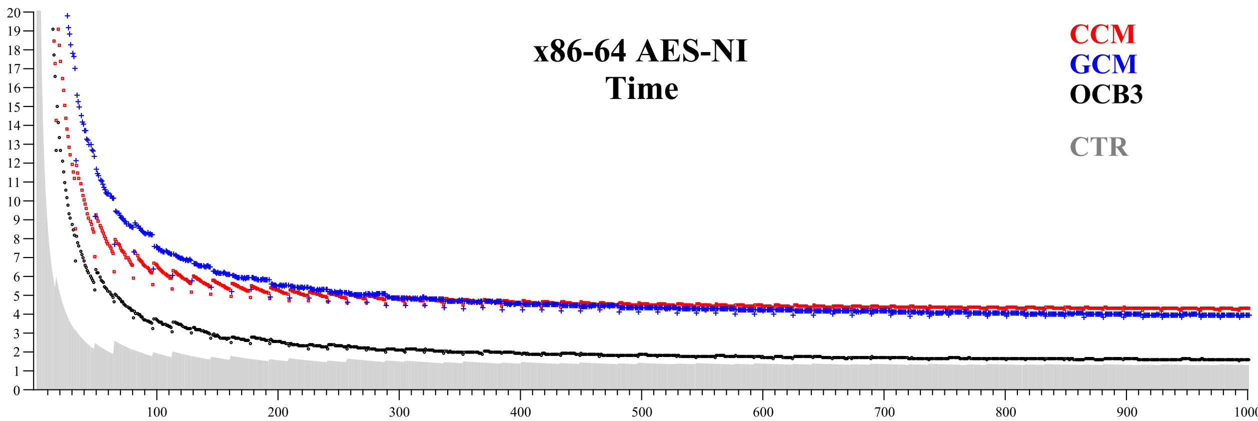

Figure 2: **Performance of CCM, GCM, and OCB3 on an x86 with AES-NI.** The x-coordinate is the message length, in bytes; the y-coordinate is the measured number of cycles per byte. From top-to-bottom on the right-hand side, the curves are for CCM, GCM, and OCB3. The shaded region shows the time for CTR mode. This and subsequent graphs are best viewed in color.

introduce something like a $GF(2^{128})$ multiply to compensate for the usually-eliminated blockcipher call, and no significant penalty is paid, compared to OCB1, if the provided nonce is not a counter (one just fails to save the blockcipher call). We go on to eliminate the latency that used to occur when computing the "checksum" and processing the AD (associated data).

Next we study the relative software performance of CCM, GCM, and the different versions of OCB. We employ the fastest publicly available code for Intel x86, both with and without Intel's new instructions for accelerating AES and GCM. For other platforms—ARM, PowerPC, and SPARC—we use a refined and popular library, OpenSSL. We test the encryption speed on messages of every byte length from 1 byte to 1 Kbyte, plus selected lengths beyond. The OCB code is entirely in C, except for a few lines of inline assembly on ARM and compiler intrinsics to access byteswap, trailing-zero count, and SSE/AltiVec functionality.

We find that, across message lengths and platforms, OCB, in any variant, is well faster than CCM and GCM. While the performance improvements from our refining OCB are certainly measurable, those differences are comparatively small. Contrary to McGrew and Viega's findings, the speed differences we observe between GCM and OCB1 are large and favor OCB1.

As an example of our experimental findings, for 4 KB messages on an Intel i5 ("Clarkdale") processor, we clock CCM at 4.17 CPU cycles per byte (cpb), GCM at 3.73 cpb, OCB1 at 1.48 cpb, OCB2 at 1.80 cpb, and OCB3 at 1.48 cpb. As a baseline, CTR mode runs at 1.27 cpb. See Figures 2 and 3. These implementations exploit the processor's AES New Instructions (AES-NI), including "carryless multiplication" for GCM. The OCB3 *authentication overhead*—the time the mode spends in excess of the time to encrypt with CTR—is about 0.2 cpb, and the difference between OCB and GCM overhead is about a factor of 10. Even written in C, our OCB implementations provide, on this platform, the fastest reported times for AE.

The means for refining OCB are not complex, but it took much work to understand what optimization would and would not help. First we wanted to arrange that nonces agreeing on all but their last few bits be processed using the same blockcipher call. To accomplish this in a way that minimizes runtime state and key-setup costs, we introduce a new hash-function family, a *stretch-then-shift* xor-universal hash. The latency reductions are achieved quite differently, by changes in how the mode defines and operates on the Checksum. Further structural changes improve support for incremental APIs.

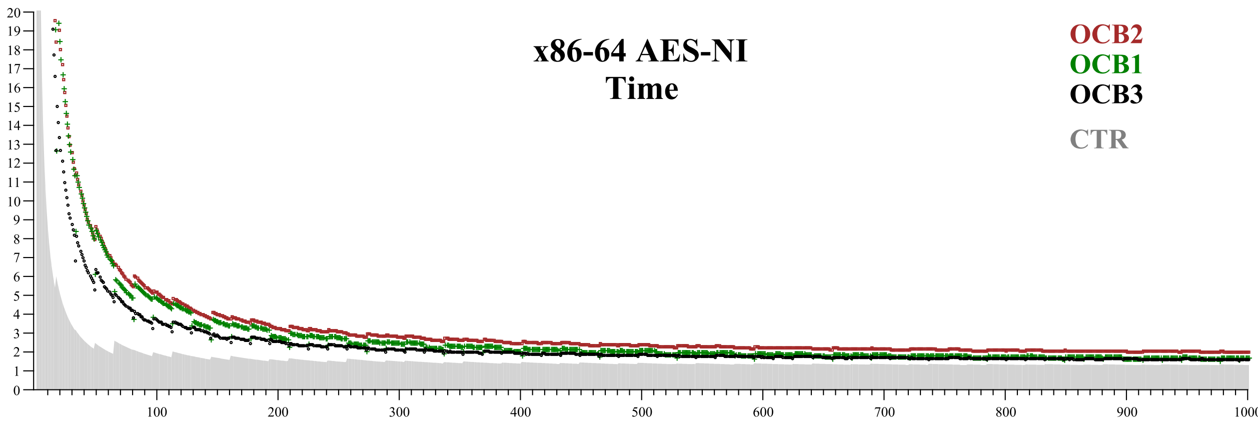

Figure 3: **Performance of OCB variants on an x86 with AES-NI.** From top-to-bottom, the curves are for OCB2, OCB1, and OCB3. The shaded region shows the time for CTR mode.

One surprising finding is that, on almost all platforms, OCB2 is slightly slower than OCB1. To explain, recall that most integrated schemes (all of Figure 1 except for RPC) involve computing an *offset* for each blockcipher call. With OCB1, each offset is computed by xoring a key-dependent value, an approach going back to Jutla [21]; with OCB2, each offset is computed by a "doubling" in GF(2^{128}). The former approach turns out to be faster. The finding emphasizes the utility of doing implementation work alongside mode design—the approach adopted for OCB3.

During our work we investigated novel ways to realize a maximal period, software-efficient, 128-bit LFSR; such constructions can also be used to make the needed offsets. A computer-aided search identified constructions like $A \,\|\, B \,\|\, C \,\|\, D \mapsto C \,\|\, D \,\|\, B \,\|\, ((A \lll 1) \oplus (A \ggg 1) \oplus (D \lll 15))$; see Appendix B. Here $|A| = |B| = |C| = |D| = 32$. While very fast, such maps are still slower than xoring a precomputed value. Our findings thus concretize Chakraborty and Sarkar's suggestion [6] to improve OCB using a fast, 128-bit, word-oriented LFSR—but, in the end, we conclude that the idea doesn't really help. Of course software-optimized 128-bit LFSRs may have other applications.

All code and data used in this paper, plus a collection of clickable tables and graphs, are available from the second author's webpage.

2 The Mode OCB3

Preliminaries. We begin with a few basics. A *blockcipher* is a deterministic algorithm $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ where $\mathcal{K}$ is a finite set and $n \geq 1$ is a number, the *key space* and *blocklength*. We require $E_K(\cdot) = E(K, \cdot)$ be a permutation for all $K \in \mathcal{K}$. Let $D = E^{-1}$ be the map from $\mathcal{K} \times \{0,1\}^n$ to $\{0,1\}^n$ defined by $D_K(Y) = D(K, Y)$ being the unique point X such that $E_K(X) = Y$.

Following recent formalizations [1,23,33,35], a scheme for (nonce-based) authenticated encryption (with associated-data) is a three-tuple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The *key space* $\mathcal{K}$ is a finite, nonempty set. The *encryption algorithm* $\mathcal{E}$ takes in a *key* $K \in \mathcal{K}$, a *nonce* $N \in \mathcal{N} \subseteq \{0,1\}^*$, a *plaintext* $M \in \mathcal{M} \subseteq \{0,1\}^*$, and *associated data* $A \in \mathcal{A} \subseteq \{0,1\}^*$. It returns, deterministically, either a ciphertext $C = \mathcal{E}_K^{N,A}(M) \in \mathcal{C} \subseteq \{0,1\}^*$ or the distinguished value Invalid. Sets $\mathcal{N}$, $\mathcal{M}$, $\mathcal{C}$, and $\mathcal{A}$ are called the *nonce space*, *message space*, *ciphertext space*, and *AD space* of Π. The *decryption algorithm* $\mathcal{D}$ takes a tuple $(K, N, A, C) \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C}$ and returns, deterministically, either Invalid or a string $M = \mathcal{D}_K^{N,A}(C) \in \mathcal{M} \subseteq \{0,1\}^*$. We require that $\mathcal{D}_K^{N,A}(C) = M$ for any string $C = \mathcal{E}_K^{N,A}(M)$ and that $\mathcal{E}$ and $\mathcal{D}$ return Invalid if provided an input outside of $\mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ or

$\mathcal{K}\times\mathcal{N}\times\mathcal{A}\times\mathcal{C}$, respectively. We require $|\mathcal{E}_K^{N,A}(M)| = |\mathcal{E}_K^{N,A}(M')|$ when the encryptions are strings and $|M| = |M'|$. If this value is always $|M| + \tau$ we call τ the *tag length* of the scheme.

DEFINITION OF OCB3. Fix a blockcipher $E\colon \mathcal{K} \times \{0,1\}^{128} \to \{0,1\}^{128}$ and a tag length $\tau \in [0..128]$. In Figure 4 we define from E and τ the AE scheme $\Pi = \text{OCB3}[E,\tau] = (\mathcal{K},\mathcal{E},\mathcal{D})$. The nonce space $\mathcal{N}$ is the set of all binary strings with fewer than 128 bits.[2] The message space $\mathcal{M}$ and AD-space $\mathcal{A}$ are all binary strings. The ciphertext space $\mathcal{C}$ is the set of all strings whose length is at least τ bits. Figure 4's procedure Setup is implicitly run on or before the first call to $\mathcal{E}$ or $\mathcal{D}$. The variables it defines are understood to be global. In the protocol definition we write $\text{ntz}(i)$ for the number of trailing zeros in the binary representation of positive integer i (eg, $\text{ntz}(1) = \text{ntz}(3) = 0$, $\text{ntz}(4) = 2$), we write $\text{msb}(X)$ for the first (most significant) bit of X, we write $A \wedge B$ for the bitwise-and of A and B, and we write $A \ll i$ for the shift of A by i positions to the left (maintaining string length, leftmost bits falling off, zero-bits entering at the right). At lines 111 and 311 we regard Bottom as a number instead of a string.

In Figure 5 we illustrate the mode. Functions Init and Inc implicitly depend on K. $\text{Init}(N)$ has the functionality corresponding to lines 106–111 of Figure 4. The other maps are simpler, with $\text{Inc}_i(\Delta) = \Delta \oplus L[\text{ntz}(i)]$, $\text{Inc}_\$(\Delta) = \Delta \oplus L_\$$, $\text{Inc}_*(\Delta) = \Delta \oplus L_*$, and $\text{Init} = 0^{128}$. Here $L_* = E_K(0^{128})$, $L_\$ = \mathbf{2} \cdot L_* = \text{double}(L_*)$, and $L[i] = \mathbf{2}^{2+i} \cdot L_*$ for all $i \geq 0$, the multiplication in $\text{GF}(2^{128})$. Value $\mathbf{2} = 0^{126}10 = \mathtt{x}$ is a particular point of the finite field.

DESIGN RATIONALE. We now explain some of the design choices made for OCB3. While not a large departure from OCB1 or OCB2, the refinements do help.

Trimming a blockcipher call. OCB1 and OCB2 took $m + 2$ blockcipher calls to encrypt an m-block string M: one to map the nonce N into an initial offset Δ; one for each block of M; one to encipher the final Checksum. The first of these is easy to eliminate if one is willing to replace the $E_K(N)$ computation by, say, $K_1 \cdot N$, the product in $\text{GF}(2^{128})$ of nonce N and a variant K_1 of K. The idea has been known since Halevi [14]. But such a change would necessitate implementing a $\text{GF}(2^{128})$ multiply for just this one step. Absent hardware support, one would need substantial precomputation and enlarged internal state to see any savings; not a net win. We therefore compute the initial offset Δ using a different xor-universal hash function: $\Delta = H_K(N) = (\text{Stretch} \ll \text{Bottom})[1..128]$ where Bottom is the last six bits of N and the $(128+64)$-bit string Stretch is made by a process involving enciphering N with its last six bits zeroed out. This *stretch-then-shift* hash will be proven xor-universal in Section 4.1. Its use ensures that, when the nonce N is a counter, the initial offset Δ can be computed without a new blockcipher call $63/64 \approx 98\%$ of the time. In this way we reduce cost from $m + 2$ blockcipher calls to an amortized $m + 1.016$ blockcipher calls, plus tiny added time for the hash.

Reduced latency. Assume the message being encrypted is not a multiple of 128 bits; there is a final block M_* having 1–127 bits. In prior versions of OCB one would need to wait on the penultimate blockcipher call to compute the Checksum and, from it, the final blockcipher call. Not only might this result in pipeline stalls [31], but if the blockcipher's efficient implementation needs a long string to ECB, then the lack of parallelizability translates to extra work. For example, Käsper and Schwabe's bit-sliced AES [22] ECB-encrypts eight AES blocks in a single shot. Using this in OCB1 or OCB2 would result in enciphering 24 blocks to encrypt a 100-byte string—three times more than what "ought" to be needed—since twice one must wait on AES output to form the next AES input. In OCB3 we restructure the algorithm so that the Checksum never depends on

[2] In practice one would either restrict nonces to byte strings of 1–15 bytes, or else demand that nonces have a fixed length, say exactly 12-bytes. Under RFC 5116, a conforming AE scheme *should* use a 12-byte nonce.

algorithm $\mathcal{E}_K^{N\,A}(M)$
if $|N| \geq 128$ **then return** Invalid
$M_1 \cdots M_m \, M_* \leftarrow M$ where each
  $|M_i| = 128$ and $|M_*| < 128$
Checksum $\leftarrow 0^{128}$; $C \leftarrow \varepsilon$
Nonce $\leftarrow 0^{127-|N|} \; 1 \; N$
Top $\leftarrow$ Nonce $\wedge 1^{122}\, 0^6$
Bottom $\leftarrow$ Nonce $\wedge 0^{122}\, 1^6$
Ktop $\leftarrow E_K$(Top)
Stretch $\leftarrow$ Ktop $\|$ (Ktop $\oplus$ (Ktop $\ll 8$))
$\Delta \leftarrow$ (Stretch $\ll$ Bottom)[1..128]
for $i \leftarrow 1$ **to** m **do**
  $\Delta \leftarrow \Delta \oplus L[\mathrm{ntz}(i)]$
  $C \overset{\|}{\leftarrow} E_K(M_i \oplus \Delta) \oplus \Delta$
  Checksum $\leftarrow$ Checksum $\oplus M_i$
if $M_* \neq \varepsilon$ **then**
  $\Delta \leftarrow \Delta \oplus L_*$
  Pad $\leftarrow E_K(\Delta)$
  $C \overset{\|}{\leftarrow} M_* \oplus \mathrm{Pad}[1\,..\,|M_*|]$
  Checksum $\leftarrow$ Checksum $\oplus M_*\, 10^*$
$\Delta \leftarrow \Delta \oplus L_\$$
Final $\leftarrow E_K$(Checksum $\oplus \Delta$)
Auth $\leftarrow \mathrm{Hash}_K(A)$
Tag $\leftarrow$ Final $\oplus$ Auth
$T \leftarrow \mathrm{Tag}[1\,..\,\tau]$
return $C \,\|\, T$

algorithm Setup(K)
$L_* \leftarrow E_K(0^{128})$
$L_\$ \leftarrow \mathrm{double}(L_*)$
$L[0] \leftarrow \mathrm{double}(L_\$)$
for $i \leftarrow 1, 2, \cdots$ **do** $L[i] \leftarrow \mathrm{double}(L[i-1])$
return

algorithm double(X)
return $(X \ll 1) \oplus (\mathrm{msb}(X) \cdot 135)$

algorithm $\mathcal{D}_K^{N\,A}(\mathcal{C})$
if $|N| \geq 128$ **or** $|\mathcal{C}| < \tau$ **then return** Invalid
$C_1 \cdots C_m \, C_* \, T \leftarrow \mathcal{C}$ where each
  $|C_i| = 128$ and $|C_*| < 128$ and $|T| = \tau$
Checksum $\leftarrow 0^{128}$; $M \leftarrow \varepsilon$
Nonce $\leftarrow 0^{127-|N|} \; 1 \; N$
Top $\leftarrow$ Nonce $\wedge 1^{122}\, 0^6$
Bottom $\leftarrow$ Nonce $\wedge 0^{122}\, 1^6$
Ktop $\leftarrow E_K$(Top)
Stretch $\leftarrow$ Ktop $\|$ (Ktop $\oplus$ (Ktop $\ll 8$))
$\Delta \leftarrow$ (Stretch $\ll$ Bottom)[1..128]
for $i \leftarrow 1$ **to** m **do**
  $\Delta \leftarrow \Delta \oplus L[\mathrm{ntz}(i)]$
  $M \overset{\|}{\leftarrow} D_K(C_i \oplus \Delta) \oplus \Delta$
  Checksum $\leftarrow$ Checksum $\oplus M_i$
if $C_* \neq \varepsilon$ **then**
  $\Delta \leftarrow \Delta \oplus L_*$
  Pad $\leftarrow E_K(\Delta)$
  $M \overset{\|}{\leftarrow} M_* \leftarrow C_* \oplus \mathrm{Pad}[1\,..\,|C_*|])$
  Checksum $\leftarrow$ Checksum $\oplus M_*\, 10^*$
$\Delta \leftarrow \Delta \oplus L_\$$
Final $\leftarrow E_K$(Checksum $\oplus \Delta$)
Auth $\leftarrow \mathrm{Hash}_K(A)$
Tag $\leftarrow$ Final $\oplus$ Auth
$T' \leftarrow \mathrm{Tag}[1\,..\,\tau]$
if $T = T'$ **then return** M
    else return Invalid

algorithm $\mathrm{Hash}_K(A)$
$A_1 \cdots A_m \, A_* \leftarrow A$ where each
  $|A_i| = 128$ and $|A_*| < 128$
Sum $\leftarrow 0^{128}$
$\Delta \leftarrow 0^{128}$
for $i \leftarrow 1$ **to** m **do**
  $\Delta \leftarrow \Delta \oplus L[\mathrm{ntz}(i)]$
  Sum $\leftarrow$ Sum $\oplus E_K(A_i \oplus \Delta)$
if $A_* \neq \varepsilon$ **then**
  $\Delta \leftarrow \Delta \oplus L_*$
  Sum $\leftarrow$ Sum $\oplus E_K(A_*\, 10^* \oplus \Delta)$
return Sum

Figure 4: **Definition of OCB3[E,τ].** Here $E\colon \mathcal{K} \times \{0,1\}^{128} \to \{0,1\}^n$ is a blockcipher and $\tau \in [0\,..\,128]$ is the tag length. Algorithms $\mathcal{E}$ and $\mathcal{D}$ are called with arguments $K \in \mathcal{K}$, $N \in \{0,1\}^{\leq 127}$, and $M, C \in \{0,1\}^*$.

any ciphertext. Concretely, Checksum $= M_1 \oplus M_2 \oplus M_{m-1} \oplus M_m 10^*$ for a short final block, and Checksum $= M_1 \oplus M_2 \oplus M_{m-1} \oplus M_m$ for a full final block. The fact that you can get the same Checksum for distinct final blocks is addressed by using different offsets in these two cases.

Incrementing offsets. In OCB1, each noninitial offset is computed from the prior one by xoring some key-derived value; the ith offset is constructed by $\Delta \leftarrow \Delta \oplus L[\mathrm{ntz}(i)]$. In OCB2, each noninitial offset is computed from the prior one by multiplying it, again in GF(2^{128}), by a constant: $\Delta \leftarrow (\Delta \ll 1) \oplus (\mathrm{msb}(\Delta) \cdot 135)$, an operation that has been called *doubling*. Not having to go to memory or

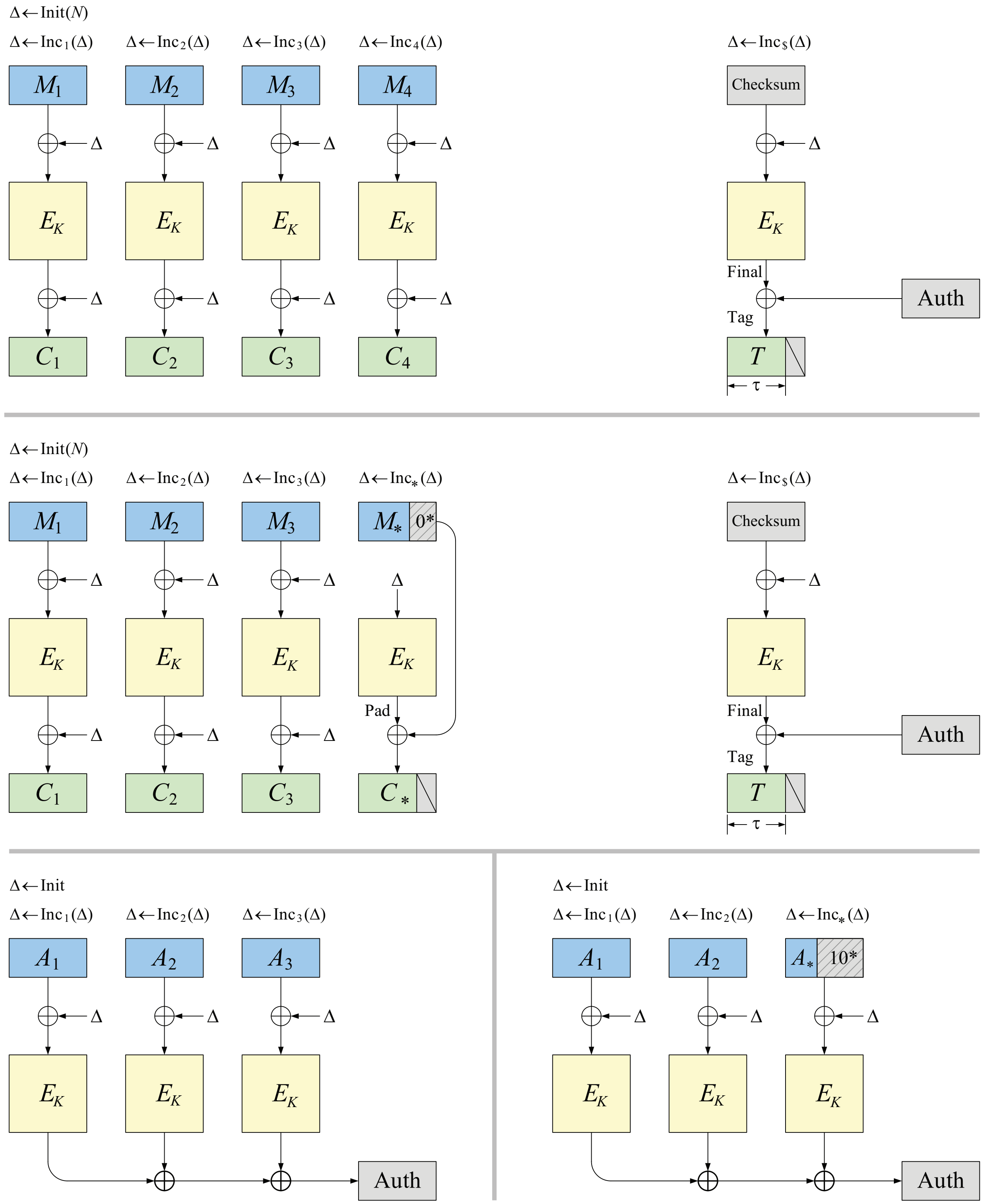

Figure 5: **Illustration of OCB3**$[E, \tau]$**.** Again $E\colon \mathcal{K} \times \{0,1\}^{128} \to \{0,1\}^n$ and $\tau \in [0\,..\,128]$. **Top:** Message M has a full final block ($|M_4| = n$) (Checksum $= M_1 \oplus M_2 \oplus M_3 \oplus M_4$). **Middle:** Message M has a short final block, $1 \le |M_*| < n$ (Checksum $= M_1 \oplus M_2 \oplus M_3 \oplus M_* 10^*$). **Bottom:** An AD of three full blocks (left) or two full blocks and one short one (right). **Throughout:** Offsets (the Δ-values) are updated and used top-to-bottom, then left-to-right. Offset initialization and update functions (Init, Inc_i, $\mathrm{Inc}_\$$, Inc_*) return n-bit strings. Each flavor of increment is an xor with some precomputed, K-dependent value.

attend to the index i, doubling was thought to be faster than the first method. In our experiments, it is not. While doubling can be coded in five Intel x86-64 assembly instructions, it still runs more slowly. In some settings, doubling loses big: it is expensive on 32-bit machines, and some compilers do poorly at turning C/C++ code for doubling into machine code that exploits the available instructions. On Intel x86, the 128-bit SSE registers lack the ability to be efficiently shifted one position to the left. Finally, the doubling operation is not endian neutral: if we must create a bit pattern in memory to match the sequence generated by doubling (and AES implementations generally do expect their inputs to live in memory) we will effectively favor big-endian architectures. We can trade this bias for a little-endian one by redefining double() to include a byteswap. But one is still favoring one endian convention over the other, and not just at key-setup time. See Appendix B for some of the alternatives to repeated doubling that we considered.

Further design issues. Unlike OCB1 and OCB2, each 128-bit block of plaintext is now processed in the same way whether or not it is the final 128 bits. This change facilitates implementing a clean incremental API, since one is able to output each 128-bit chunk of ciphertext after receiving the corresponding chunk of plaintext, even if it is not yet known if the plaintext is complete.

All AD blocks can now be processed concurrently; in OCB2, the penultimate block's output was needed to compute the final block's input, potentially creating pipeline stalls or inefficient use of a blockcipher's multi-block ECB interface. Also, each 128-bit block of AD is treated the same way if it is or isn't the message's end, simplifying the incremental provisioning of AD.

We expect the vast majority of processors running OCB3 will be little-endian; still, the mode's definition does nothing to favor this convention. The issue arises each time "register oriented" and "memory oriented" values interact. These are the same on big-endian machines, but are opposite on little-endian ones. One could, therefore, favor little-endian machines by building into the algorithm byte swaps that mimic those that would occur naturally each time memory and register oriented data interact. We experimentally adapted our implementation to do this but found that it made very little performance difference. This is due, first, to good byte reversal facilities on most modern processors (eg, `pshufb` can reverse 16 bytes on our x86 in a single cycle). It is further due to the fact that OCB3's table-based approach for incrementing offsets allows for the table to be endian-adjusted at key setup, removing most endian-dependency on subsequent encryption or decryption calls. Since it makes little difference to performance, and big-endian specifications are conceptually easier, OCB3 does not make any gestures toward little-endian orientation.

A low-level choice where OCB and GCM part ways is in the representation of field points. In GCM the polynomial $a_{127}\mathtt{x}^{127} + \cdots a_1\mathtt{x} + a_0$ corresponds to string $a_0 \ldots a_{127}$ rather than $a_{127} \ldots a_0$. McGrew and Viega call this the little-endian representation, but, in fact, this choice has nothing to do with endianness. The usual convention on machines of all kinds is that the msb is the left-most bit of any register. Because of this, GCM's "reflected-bit" convention can result in extra work to be performed even on Intel chips having instructions specifically intended for accelerating GCM [12, 13]. Among the advantages of following the msb-first convention is that a left shift by one can be implemented by adding a register to itself, an operation often faster than a logical shift.

SECURITY OF OCB3. First we provide our definitions. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an AE scheme. Given an adversary (algorithm) $\mathcal{A}$, we let $\mathbf{Adv}^{\mathrm{priv}}_{\Pi}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot,\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(\cdot,\cdot,\cdot)} \Rightarrow 1]$ where queries of $\$(N, A, M)$ return a uniformly random string of length $|\mathcal{E}^{N,A}_K(M)|$. We demand that $\mathcal{A}$ never asks two queries with the same first component (the N-value), that it never ask a query outside of $\mathcal{N} \times \mathcal{A} \times \mathcal{M}$, and that it never repeats a query. Next we define authenticity. For that, let $\mathbf{Adv}^{\mathrm{auth}}_{\Pi}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot,\cdot)}$ forges$]$ where we say that the adversary *forges* if it outputs a value $(N, A, C) \in \mathcal{N} \times \mathcal{A} \times \mathcal{C}$ such that $\mathcal{D}^{N,A}_K(C) \neq$ INVALID yet there was no prior query (N, A, M') that returned C. We demand that $\mathcal{A}$ never asks two queries with the same first component (the N-value), never asks a query outside of $\mathcal{N} \times \mathcal{A} \times \mathcal{M}$, and never repeats a query.

When $E\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ is a blockcipher define $\mathbf{Adv}_E^{\pm\mathrm{prp}}(\mathcal{A}) = \Pr[\mathcal{A}^{E_K(\cdot),\, E_K^{-1}(\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot),\, \pi^{-1}(\cdot)} \Rightarrow 1]$ where K is chosen uniform from $\mathcal{K}$ and $\pi(\cdot)$ is a uniform permutation on $\{0,1\}^n$. Define $\mathbf{Adv}_E^{\mathrm{prp}}(\mathcal{A}) = \Pr[\mathcal{A}^{E_K(\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot)} \Rightarrow 1]$ by removing the decryption oracle. The *ideal* blockcipher of blocksize n is the blockcipher $\mathrm{Bloc}[n]\colon \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ where each key K names a distinct permutation.

The security of OCB3 is given by the following theorem. We give the result in its information-theoretic form. Passing to the complexity-theoretic setting, where the idealized blockcipher $\mathrm{Bloc}[n]$ is replaced by a conventional blockcipher secure as a strong-PRP, is standard.

Theorem 1 *Fix $n = 128$, $\tau \in [0\,..\,n]$, and let $\Pi = \mathrm{OCB3}[E, \tau]$ where $E = \mathrm{Bloc}[n]$ is the ideal blockcipher on n bits. If $\mathcal{A}$ asks encryption queries that entail σ total blockcipher calls, then $\mathbf{Adv}_\Pi^{\mathrm{priv}}(\mathcal{A}) \le 6\,\sigma^2/2^n$. Alternatively, if $\mathcal{A}$ asks encryption queries then makes a forgery attempt that together entail σ total blockcipher calls, then $\mathbf{Adv}_\Pi^{\mathrm{auth}}(\mathcal{A}) \le 6\sigma^2/2^n + (2^{n-\tau})/(2^n - 1)$.* □

When we speak of the number of blockcipher calls entailed we are adding up the (rounded-up) blocklength for all the different strings output by the adversary and adding in $q+2$ (q =number of queries), to upper-bound blockcipher calls for computing L_* and the initial Δ values. Main elements of the proof are described in Section 4.

3 Experimental Results

Scope and codebase. We empirically study the software performance of OCB3, and compare this with state-of-the-art implementations of GCM, which delivers the fastest previously reported AE times. Both modes are further compared against CTR, the fastest privacy-only mode, which makes a good baseline for answering how much extra one pays for authentication. Finally, we consider CCM, the first NIST-approved AE scheme, and also OCB1 and OCB2, which are benchmarked to show how the evolution of OCB has affected performance.

Intensively optimized implementations of CTR and GCM are publicly available for the x86. Käsper and Schwabe hold the speed record for 64-bit code with no AES-NI, reporting peak rates of 7.6 and 10.7 CPU cycles per byte (cpb) for CTR and GCM [22]. With AES-NI, developmental versions of OpenSSL achieve 1.3 cpb for CTR [32] and 3.3 cpb for GCM.[3] These various results use different x86 chips and timing mechanisms. Here we use the Käsper-Schwabe AES, CTR, and GCM, the OpenSSL CTR, CCM, and GCM, augment the collection with new code for OCB, and compare performance on a single x86 and use a common timing mechanism, giving the fairest comparison to date.

The only non-proprietary, architecture-specific non-x86 implementations for AES and GCM that we could find are those in OpenSSL. Although these implementations are hand-tuned assembly, they are designed to be timing-attack resistant, and are therefore somewhat slow. This does not make comparisons with them irrelevant. OCB is timing-attack resistant too (assuming the underlying blockcipher is), making the playing field level. We adopt the OpenSSL implementations for non-x86 comparisons and emphasize that timing-resistant implementations are being compared, not versions written for ultimate speed.

[3] Andy Polyakov, personal communication, August 27, 2010. The fastest published AES-NI time for GCM is 3.5 cpb on 8KB messages, from Gueron and Kounavis [13].

The OCB1 and OCB2 implementations are modifications of our OCB3 implementation, and therefore are similarly optimized. These implementations are in C, calling out to AES. No doubt further performance improvements can be obtained by rewriting the OCB code in assembly.

Hardware and software environments. We selected five representative instruction-set architectures: (1) 32-bit x86, (2) 64-bit x86, (3) 32-bit ARM, (4) 64-bit PowerPC, and (5) 64-bit SPARC. Collectively, these architectures dominate the workstation, server, and portable computing marketplace. The x86 processor used for both 32- and 64-bit tests is an Intel Core i5-650 "Clarkdale" supporting the AES-NI instructions. The ARM is a Cortex-A8. The PowerPC is a 970fx. The SPARC is an UltraSPARC IIIcu. Each runs Debian Linux 6.0 with kernel 2.6.35 and GCC 4.5.1. Compilation is done with `-O3` optimization, `-mcpu` or `-march` set according to the host processor, and `-m64` to force 64-bit compilation when needed.

Testing methodology. The number of CPU cycles needed to encrypt a message is divided by the length of the message to arrive at the cost per byte to encrypt messages of that length. This is done for every message length from 1 to 1024 bytes, as well as 1500 and 4096 bytes. So as not to have performance results overly influenced by the memory subsystem of a host computer, we arrange for all code and data to be in level-1 cache before timing begins. Two timing strategies are used: C clock and x86 time-stamp counter. In the clock version, the ANSI C `clock()` function is called before and after repeatedly encrypting the same message, on sequential nonces, for a little more than one second. The clock difference determines how many CPU cycles were spent on average per processed byte. This method is highly portable, but it is time-consuming when collecting an entire dataset. On x86 machines there is a "time-stamp counter" (TSC) that increments once per CPU cycle. To capture the average cost of encryption—including the more expensive OCB3 encryptions that happen once every 64 calls—the TSC is used to time encryption of the same message 64 times on successive counter-based nonces. The TSC method is not portable, working only on x86, but is fast. Both methods have their potential drawbacks. The clock method depends on the hardware having a high-resolution timer and the OS doing a good job of returning the time used only by the targeted process. The TSC read instruction might be executed out of order, in some cases it has high latency, and it continues counting when other processes run.[4] In the end, we found that both timing methods give similar results. For example, in the eighteen x86 test runs done for this paper, the Internet Performance Index values computed by the two methods varied by no more than 0.05 cpb 10 times, no more than 0.10 cpb 15 times, and no more than 0.20 cpb all 18 times.

Results. Summary findings are presented in Figures 2, 3, and 6. On all architectures and message lengths, OCB3 is significantly faster than GCM and CCM. Except on very short messages, it is nearly as fast as CTR. On x86, GCM's most competitive platform, OCB3's authentication overhead (its cost beyond CTR encryption) is 4–16%, with or without AES-NI, on both an Internet Performance Index (IPI)[5] and 4KB message length basis. In all our tests, CCM never has IPI or 4KB rates better than GCM, coming close only when small registers make GCM's multiplications expensive, or AES-NI instructions speed CCM's block encipherments. Results are similar on other

[4] To lessen these problems we read the TSC once before and after encrypting the same message 65 times, then read the TSC once before and after encrypting the same message once more. Subtracting the second timing from the first gives us the cost for encrypting the message 64 times, and mitigates the out-of-order and latency problems. To avoid including context-switches, we run experiments multiple times and keep only the median timing.

[5] The IPI is a weighted average of timings for messages of 44 bytes (5%), 552 bytes (15%), 576 bytes (20%), and 1500 bytes (60%) [31]. It is based on Internet backbone studies from 1998. We do not suggest that the IPI reflects a contemporary, real-world distribution of message lengths, only that it is useful to have *some* metric that attends to shorter messages and those that are not a multiple of 16 bytes. Any metric of this sort will be somewhat arbitrary in its definition.

x86-64 AES-NI				
Mode	T_{4K}	T_{IPI}	Size	Init
CCM	4.17	4.57	512	265
GCM	3.73	4.53	656	337
OCB1	1.48	2.08	544	251
OCB2	1.80	2.41	448	185
OCB3	1.48	1.87	624	253
CTR	1.27	1.37	244	115

x86-32 AES-NI				
Mode	T_{4K}	T_{IPI}	Size	Init
CCM	4.18	4.70	512	274
GCM	3.88	4.79	656	365
OCB1	1.60	2.22	544	276
OCB2	1.79	2.42	448	197
OCB3	1.59	2.04	624	270
CTR	1.39	1.52	244	130

x86-64 Käsper-Schwabe				
Mode	T_{4K}	T_{IPI}	Size	Init
GCM	22.4	26.7	1456	3780
GCM-8K	10.9	15.2	9648	2560
OCB1	8.28	13.4	3008	3390
OCB2	8.55	13.6	2912	3350
OCB3	8.05	9.24	3088	3480
CTR	7.74	8.98	1424	1180

ARM Cortex-A8				
Mode	T_{4K}	T_{IPI}	Size	Init
CCM	51.3	53.7	512	1390
GCM	50.8	53.9	656	1180
OCB1	29.3	31.5	672	1920
OCB2	28.5	31.8	576	1810
OCB3	28.9	30.9	784	1890
CTR	25.4	25.9	244	236

PowerPC 970				
Mode	T_{4K}	T_{IPI}	Size	Init
CCM	75.7	77.8	512	1510
GCM	53.5	56.2	656	1030
OCB1	38.2	41.0	672	2180
OCB2	38.1	41.1	576	2110
OCB3	37.5	39.6	784	2240
CTR	37.5	37.8	244	309

UltraSPARC III				
Mode	T_{4K}	T_{IPI}	Size	Init
CCM	49.4	51.7	512	1280
GCM	39.3	41.5	656	904
OCB1	25.5	27.7	672	1720
OCB2	24.8	27.0	576	1700
OCB3	25.0	26.5	784	1730
CTR	24.1	24.4	244	213

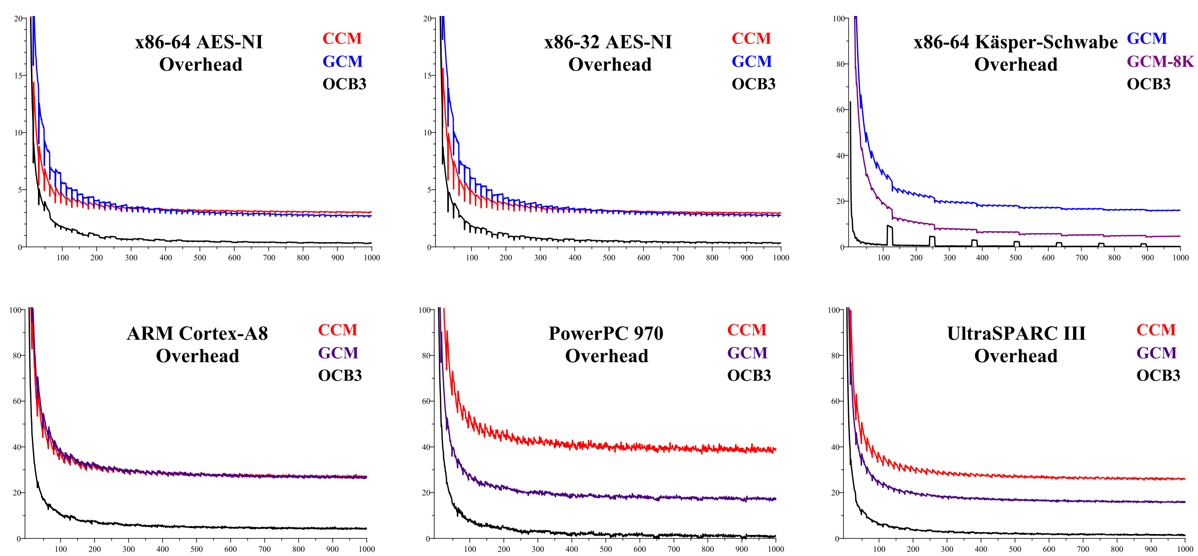

Figure 6: **Empirical performance of AE modes.** For each architecture we give time to encrypt 4KB messages (in CPU cycles per byte), time to encrypt a weighted basket of message lengths (IPI, also in cpb), size of the implementation's context (in bytes), and time to initialize key-dependent values (in CPU cycles). Next we graph the same data, subtracting the CTR time and dropping the curves for OCB1 and OCB2, which may be visually close to that of OCB3. The CCM and GCM curves are visually hard to distinguish in the x86-64 AES NI, x86-32 AES NI, and ARM Cortex-A8 graphs.

architectures. The overhead of OCB3 does not exceed 12% that of GCM or CCM on PowerPC or SPARC, or 18% on ARM, when looking at either IPI or 4KB message encryption rates.

To see why OCB3 does so well, consider that there are four phases in OCB3 encryption: initial offset generation, encryption of full blocks, encryption of a partial final block (if there is one), and tag generation. On all but the shortest messages, full-block processing dominates overall cost per byte. Here OCB3, and OCB1, are particularly efficient. An unrolled implementation of, say, four blocks per iteration, will have, as overhead on top of the four blockcipher calls and the reads and writes associated to them: 16 xor operations (each on 16-byte words), 1 ntz computation, and 1 table lookup of a 16-byte value. On x86, summing the latencies of these 18 operations—which ignores the potential for instruction-level parallelism (ILP)—the operations require 23 cycles, or 0.36 cpb. In reality, on 64-bit x64 using AES-NI, we see CTR taking 1.27 cpb on 4KB messages while OCB3 uses 1.48, an overhead of 0.21 cpb, the savings coming from the ILP.

Short messages are optimized for too. When there is little or no full-block processing, it is the other three phases of encryption that determine performance. One gets a sense of the cost of these—initial offset generation, encryption of a partial final block, and tag generation—by looking at the cost to encrypt a single byte. On x86, OpenSSL's AES-NI based CTR implementation does this in 86 cycles, while CCM, GCM, and OCB3 use 257, 354, and 249 cycles, respectively. CCM remains competitive with OCB3 only for very short strings. On 64-bit x86 without AES-NI, using Käsper-Schwabe's bit-sliced AES that processes eight blocks at once, OCB3's performance lead is much greater, as its two blockcipher calls can be computed concurrently, unlike CCM and GCM. In this scenario, single-byte encryption rates for CCM, GCM, OCB3, CTR are 2600, 2230, 1080, 1010 cycles. On the other three architectures we see the following single-byte encryption times for (CCM, GCM, OCB3; CTR): ARM (1770, 1950, 1190; 460), PowerPC (2520, 1860, 1450; 309), and SPARC (1730, 1520, 1770; 467).

With hardware support making AES very cheap, authentication overhead becomes more prominent. AES-NI instructions enable AES-128 throughput of around 20 cycles per block. VIA's `xcrypt` assembly instruction is capable of 10 cycles per block on long ECB sequences [38]. Speeds like these can make authentication overhead more expensive than encryption. With the Käsper-Schwabe code (no AES-NI), for example, on an IPI basis, OCB3 overhead is only 3% of encryption cost, but under AES-NI it rises to 27%. Likewise, GCM overhead rises from 41% to 70%. One might think CCM would do well using AES-NI since its overhead is mostly blockcipher calls, but its use of (serial) CBC for authentication reduces AES throughput to around 60 cycles per block, causing authentication overhead of about 70%.[6]

A processor with AES-NI provides a nearly ideal environment for OCB3: there are sixteen 16-byte registers available for caching recently used values, performing xor operations, and these registers also provide the interface for AES calls. The assembly produced by GCC for the full-block processing loop was able to keep all values in registers except for AES round keys, resulting in exceptional performance. When not using AES-NI on x86, overhead increases slightly due to function-call overhead and the use of a memory-based interface for AES. On 64-bit x86 using Käsper-Schwabe's AES implementation, OCB3 costs 0.36 cpb more than CTR. We see similar results on PowerPC and SPARC. OCB3 has higher overhead on ARM due to its small register set, but still has an overhead 1/7th that of GCM.

[6] Intel released their Sandy Bridge microarchitecture January 2011, too late for a thorough update of this paper. Sandy Bridge increases both AES throughput and latency. Under Sandy Bridge, OCB and CTR will be substantially faster (likely under 1.0 cpb on long messages) because their work is dominated by parallel AES invocations. GCM will be just a little faster because most of its time is spent in authentication, which does not benefit from Sandy Bridge. CCM will be slower because longer latencies negatively affect CBC authentication.

As expected, OCB1 and OCB3 long-message performance is the same due to having identical full-block processing. OCB2 is slower on long messages on all tested platforms but SPARC (computing ntz is slow on SPARC). With a counter-based nonce, OCB3 computes its initial encryption offset using a few bitwise shifts of a cached value rather than generating it with a blockcipher as both OCB1 and OCB2 do. This results in significantly improved average performance for encryption of short messages. The overall effect is that on an IPI basis on, say, 64-bit x86 using AES-NI, OCB3's authentication overhead is only 65% of that for OCB1 and only 40% of that for OCB2. When the provided nonce is *not* a counter, OCB3 performance is, in most of our test environments, indistinguishable from that of OCB1.

4 Proof of Security for OCB3

We describe three elements in the proof of OCB3's security: (1) the new xor-universal hash function it employs; (2) the definition and proof for a simple TBC (tweakable blockcipher) based generalization of OCB3; and (3) the proof that the particular TBC used by OCB3 is good.

4.1 Stretch-then-Shift Universal Hash

A new hash function H underlies the mapping of the low-order bits of the nonce to a 128-bit string (lines 108, 110, and 111 of Figure 4). While an off-the-shelf hash would have worked alright, we were able to do better for this step. We start with the needed definitions.

Definition. Let $\mathcal{K}$ be a finite set and let $H\colon \mathcal{K}\times\mathcal{X}\to\{0,1\}^n$ be a function. We say that H is *strongly xor-universal* if for all distinct $x, x' \in \mathcal{X}$ we have that $H_K(x)\oplus H_K(x')$ is uniformly distributed in $\{0,1\}^n$ and, also, $H_K(x)$ is uniformly distributed in $\{0,1\}^n$ for all $x\in\mathcal{X}$. The first requirement is the usual definition for H being *xor-universal*; the second we call *universal-1*.

The technique. We aim to construct strongly xor-universal hash-functions $H\colon \mathcal{K}\times\mathcal{X}\to\{0,1\}^n$ where $\mathcal{K}=\{0,1\}^{128}$, $\mathcal{X}=[0\,..\,\mathrm{domSize}-1]$, and $n=128$. We want domSize to be at least some modest-size number, say domSize ≥ 64, and intend that computing $H_K(x)$ be almost as fast as doing a table lookup. Fast computation of H should not require any large table, nor the preprocessing of K. Our desire for extreme speed in the absence of preprocessing and big tables rules out methods based on $\mathrm{GF}(2^{128})$ multiplication, the obvious first attempt.

The method we propose is to stretch the key K into a longer string *stretch*(K), and then extract its bits $x+1$ to $x+128$. Symbolically, $H_K(x)=(\mathit{stretch}(K))[x+1\,..\,x+128]$ where $S[a\,..\,b]$ denotes bits a through b of S, indexing beginning with 1. Equivalently, $H_K(x)=(\mathit{stretch}(K)\ll x)[1\,..\,128]$. We call this a *stretch-then-shift* hash.

How to stretch K? It seems natural to have *stretch*(K) begin with K, so let's assume that $\mathit{stretch}(K)=K\parallel s(K)$ for some function s. It's easy to see that $s(K)=K$ and $s(K)\ll c$ won't work, but $s(K)=K\oplus(K\ll c)$, for some constant c, looks plausible for accommodating modest-sized domain. We now demonstrate that, for well-chosen c, this function does the job.

Analysis. To review, we are considering the hash functions $H^c_K(x)=(\mathrm{Stretch}\ll x)[1\,..\,128]$ where $\mathrm{Stretch}=\mathit{stretch}(K)=K\parallel(K\oplus(K\ll c))$ and $c\in[0\,..\,127]$. We'd like to know the maximal value of domSize for which $H_K(x)$ is xor-universal on the domain $\mathcal{X}=[0\,..\,\mathrm{domSize}(c)-1]$. This can be calculated by a computer program, as we now explain. Fix c and consider the 256×128 entry matrix $A=\begin{pmatrix} I\\ J\end{pmatrix}$ where I is the 128×128 identity matrix and J is the 128×128-bit matrix for which $J_{ij}=1$ iff $j=i$ or $j=i+c$. Let A_i denote the 128×128 submatrix of A that includes

c	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
domSize(c)	3	15	7	3	124	7	3	85	120	3	118	63	3	31	63	3	7	31	3	7

Figure 7: **Stretch-then-shift hash.** Largest $\mathcal{X} = [0..\text{domSize}(c)-1]$ s.t. $H_K^c(x) = (\text{Stretch}(K) \ll x)[1..128]$ is strongly xor-universal when $c \in [1..16]$, $K \in \{0,1\}^{128}$, $x \in \mathcal{X}$, and $\text{Stretch}(K) = K \,\|\, (K \oplus (K \ll c))$.

only A's rows i to $i+127$. Then $H_K^c(x) = A_{x+1}K$, the product in GF(2) of the matrix A_{i+1} and the column vector K. Let $B_{i,j} = A_i + A_j$ be the indicated 128×128 matrix, the matrix sum over GF(2). We would like to ensure that, for arbitrary $0 \le i < j < \text{domSize}(c)$ and a uniform $K \in \{0,1\}^{128}$ that the 128-bit string $H_K^c(i) + H_K^c(j)$ is uniform—which is to say that $A_{i+1}K + A_{j+1}K = (A_{i+1} + A_{j+1})K = B_{i+1,j+1}K$ is uniform. This will be true if and only if $B_{i,j}$ is invertible in GF(2) for all $1 \le i < j \le \text{domSize}(c)$. Thus domSize($c$) can be computed as the largest number domSize(j) such that $B_{i,j}$ is full rank, over GF(2), for all $1 \le i < j \le \text{domSize}(j)$. Recalling the universal-1 property we also demand that A_i have full rank for all $1 \le i \le \text{domSize}(c)$. Now for any c, the number of matrices $A_{i,j}$ to consider is at most 2^{13}, and finding the rank in GF(2) of that many 128×128 matrices is a feasible calculation.

Our results are tabulated in Figure 7. The most interesting cases are H^5 and H^8, which are strongly xor-universal on $\mathcal{X} = [0..123]$ and $\mathcal{X} = [0..84]$, respectively. We offer no explanation for why these functions do well and various other H^c do not. As both H^5 and H^8 work on $[0..63]$ we select the latter map for use in OCB3 and single out the following result:

Lemma 1 *Let* $H\colon \{0,1\}^{128} \times [0..63] \to \{0,1\}^{128}$ *be defined by* $H_K(x) = (\text{Stretch} \ll x)[1..128]$ *where* $\text{Stretch} = K \,\|\, (K \oplus (K \ll 8))$. *Then H is strongly xor-universal.* □

Efficiency. On 64-bit computers, assuming $K \,\|\, (K \oplus (K \ll 8))$ is precomputed and in memory, the value of $H_K(x)$ can be computed by three memory loads and two multiprecision shifts, requiring fewer than ten cycles on most architectures. If only K is in memory then the first 64 bits of $K \oplus (K \ll 8)$ can be computed with three additional assembly instructions. In the absence of a preprocessed table or special hardware-support, a method based on $\text{GF}(2^{128})$ multiplies would not fare nearly as well.

Computing successive H_K^c values requires a single extended-precision shift, making stretch-then-shift a reasonable approach for incrementing offsets. Unfortunately, it is not endian-neutral.

4.2 The TBC-Based Generalization of OCB3

Following the insight of Liskov, Rivest, and Wagner [28], OCB3 can be understood as an instantiation of an AE scheme that depends on a *tweakable blockcipher* (TBC). This is a deterministic algorithm $\widetilde{E}$ having signature $\widetilde{E}\colon \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$ where $\mathcal{K}$ and $\mathcal{T}$ are sets and $n \ge 1$ is a number—the *key space*, *tweak space*, and *blocklength*, respectively. We require $\widetilde{E}_K^T(\cdot) = \widetilde{E}(K,T,\cdot)$ be a permutation for all $K \in \mathcal{K}$ and $T \in \mathcal{T}$. Write $\widetilde{D} = \widetilde{E}^{-1}$ for the map from $\mathcal{K} \times \mathcal{T} \times \{0,1\}^n$ to $\{0,1\}^n$ defined by $\widetilde{D}_K^T(Y) = \widetilde{D}(K,T,Y)$ being the unique X such that $\widetilde{E}_K^T(X) = Y$. The *ideal* TBC for a tweak set $\mathcal{T}$ and blocksize n is the blockcipher $\text{Bloc}[\mathcal{T},n]\colon \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$ where the keys name distinct permutations for each tweak T. For $\mathcal{T} = \mathcal{T}^{\pm} \cup \mathcal{T}^{+}$, $\mathcal{T}^{\pm} \cap \mathcal{T}^{+} = \emptyset$, let $\mathbf{Adv}_{\widetilde{E}}^{\text{prp}[\mathcal{T}^{\pm}]}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K}\colon \mathcal{A}^{\widetilde{E}_K(\cdot,\cdot),\, \widetilde{D}_K(\cdot,\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot,\cdot),\, \pi^{-1}(\cdot,\cdot)} \Rightarrow 1]$ where π is chosen uniformly from $\text{Bloc}[\mathcal{T},n]$ and adversary $\mathcal{A}$ is only allowed to ask decryption queries (T,Y) with $T \in \mathcal{T}^{\pm}$. Write $\mathbf{Adv}_{\widetilde{E}}^{\pm\text{prp}}(\mathcal{A})$ for $\mathbf{Adv}_{\widetilde{E}}^{\text{prp}[\mathcal{T}]}(\mathcal{A})$ and $\mathbf{Adv}_{\widetilde{E}}^{\text{prp}}(\mathcal{A})$ for $\mathbf{Adv}_{\widetilde{E}}^{\text{prp}[\emptyset]}(\mathcal{A})$. Our definition

```
algorithm E_K^{N,A}(M)
if N ∉ 𝒩 then return Invalid
M_1 ··· M_m M_* ← M where each
        |M_i| = n and |M_*| < n
Checksum ← 0^n,  C_* ← ε
for i ← 1 to m do
        C_i ← Ẽ_K^{N i}(M_i)
        Checksum ← Checksum ⊕ M_i
if M_* = ε then Final ← Ẽ_K^{N m $}(Checksum)
else  Pad ← Ẽ_K^{N m *}(0^n)
        C_* ← M_* ⊕ Pad[1 .. |M_*|]
        Checksum ← Checksum ⊕ M_* 10^*
        Final ← Ẽ_K^{N m * $}(Checksum)
Auth ← Hash_K(A)
Tag ← Final ⊕ Auth
T ← Tag[1 .. τ]
return C_1 ··· C_m C_* || T

algorithm Hash_K(A)
Sum ← 0^n
A_1 ··· A_m A_* ← A for |A_i| = n, |A_*| < n
for i ← 1 to m do
        Sum ← Sum ⊕ Ẽ_K^{i}(A_i)
if A_* ≠ ε then
        Sum ← Sum ⊕ Ẽ_K^{m *}(A_* 10^*)
return Sum
```

```
algorithm D_K^{N,A}(C)
if N ∉ 𝒩 or |C| < τ then return Invalid
C_1 ··· C_m C_* T ← C where each
        |C_i| = n, |C_*| < n, and |T| = τ
Checksum ← 0^n,  M_* ← ε
for i ← 1 to m do
        M_i ← D̃_K^{N i}(C_i)
        Checksum ← Checksum ⊕ M_i
if C_* = ε then Final ← Ẽ_K^{N m $}(Checksum)
else  Pad ← Ẽ_K^{N m *}(0^n)
        M_* ← C_* ⊕ Pad[1 .. |C_*|]
        Checksum ← Checksum ⊕ M_* 10^*
        Final ← Ẽ_K^{N m * $}(Checksum)
Auth ← Hash_K(A)
Tag ← Final ⊕ Auth
T' ← Tag[1 .. τ]
if T = T' then return M_1 ··· M_m M_*
        else return Invalid
```

Figure 8: **Definition of ΘCB3$[\widetilde{E},\tau]$.** Here $\widetilde{E}\colon \mathcal{N} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$ is a tweakable blockcipher and $\tau \in [0\,..\,n]$ is the tag length. We have that $\mathrm{OCB3}[E,\tau] = \Theta\mathrm{CB3}[\widetilde{E},\tau]$ for an appropriately chosen $\widetilde{E}$.

unifies PRP and strong-PRP security, allowing forward queries for all tweaks and backwards queries for those in $\mathcal{T}^{\pm}$. A conventional blockcipher can be regarded as a TBC with a singleton tweak space.

THE ΘCB3 SCHEME. Fix an arbitrary set of nonces $\mathcal{N}$; for concreteness, say $\mathcal{N} = \{0,1\}^{<128}$. Define from this set the corresponding tweak space $\mathcal{T}$ by way of

$$\mathcal{T} \;=\; \mathcal{N}\times\mathbb{N}_1 \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{*\} \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{\$\} \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{*\$\} \;\cup\; \mathbb{N}_1 \;\cup\; \mathbb{N}_0\times\{*\}$$

where $\mathbb{N}_1$ and $\mathbb{N}_0$ are the positive and nonnegative integers, respectively. Tweaks, it can be seen, are of six mutually exclusive "types." Tweaks of the first type are in the set $\mathcal{T}^{\pm} = \mathcal{N}\times\mathbb{N}_1$. Omitting parenthesis and commas when writing tweaks, TBC calls will look like $\widetilde{E}_K^{N\,i}(X)$, $\widetilde{E}_K^{N\,i\,*}(X)$, $\widetilde{E}_K^{N\,i\,\$}(X)$, $\widetilde{E}_K^{N\,i\,*\,\$}(X)$, $\widetilde{E}_K^{i}(X)$, or $\widetilde{E}_K^{i*}(X)$. Now given such a TBC $\widetilde{E}\colon \mathcal{K}\times\mathcal{T}\times\{0,1\}^n \to \{0,1\}^n$ and given a tag length $\tau \in [0\,..\,n]$, we construct the AE scheme $\Pi = \Theta\mathrm{CB3}[\widetilde{E},\tau] = (\mathcal{K},\mathcal{E},\mathcal{D})$ as defined in Figure 8. The scheme's nonce space is $\mathcal{N}$, the message space is $\mathcal{M} = \{0,1\}^*$, the AD space is $\mathcal{A} = \{0,1\}^*$, and the ciphertext space is $\mathcal{C} = \{0,1\}^{\geq\tau}$. The scheme is illustrated in Figure 9.

We now describe the security of ΘCB3 when using an ideal TBC. The proof is given in Appendix A.1.

Lemma 2 *Let $\Pi = \Theta\mathrm{CB3}[\widetilde{E},\tau]$ where $\widetilde{E} = \mathrm{Bloc}[\mathcal{T},n]\colon \mathcal{K}\times\mathcal{T}\times\{0,1\}^n \to \{0,1\}^n$ is ideal. Let $\mathcal{A}$ be an adversary. Then $\mathbf{Adv}_{\Pi}^{\mathrm{priv}}(\mathcal{A}) = 0$ and $\mathbf{Adv}_{\Pi}^{\mathrm{auth}}(\mathcal{A}) \leq (2^{n-\tau})/(2^n-1)$.* □

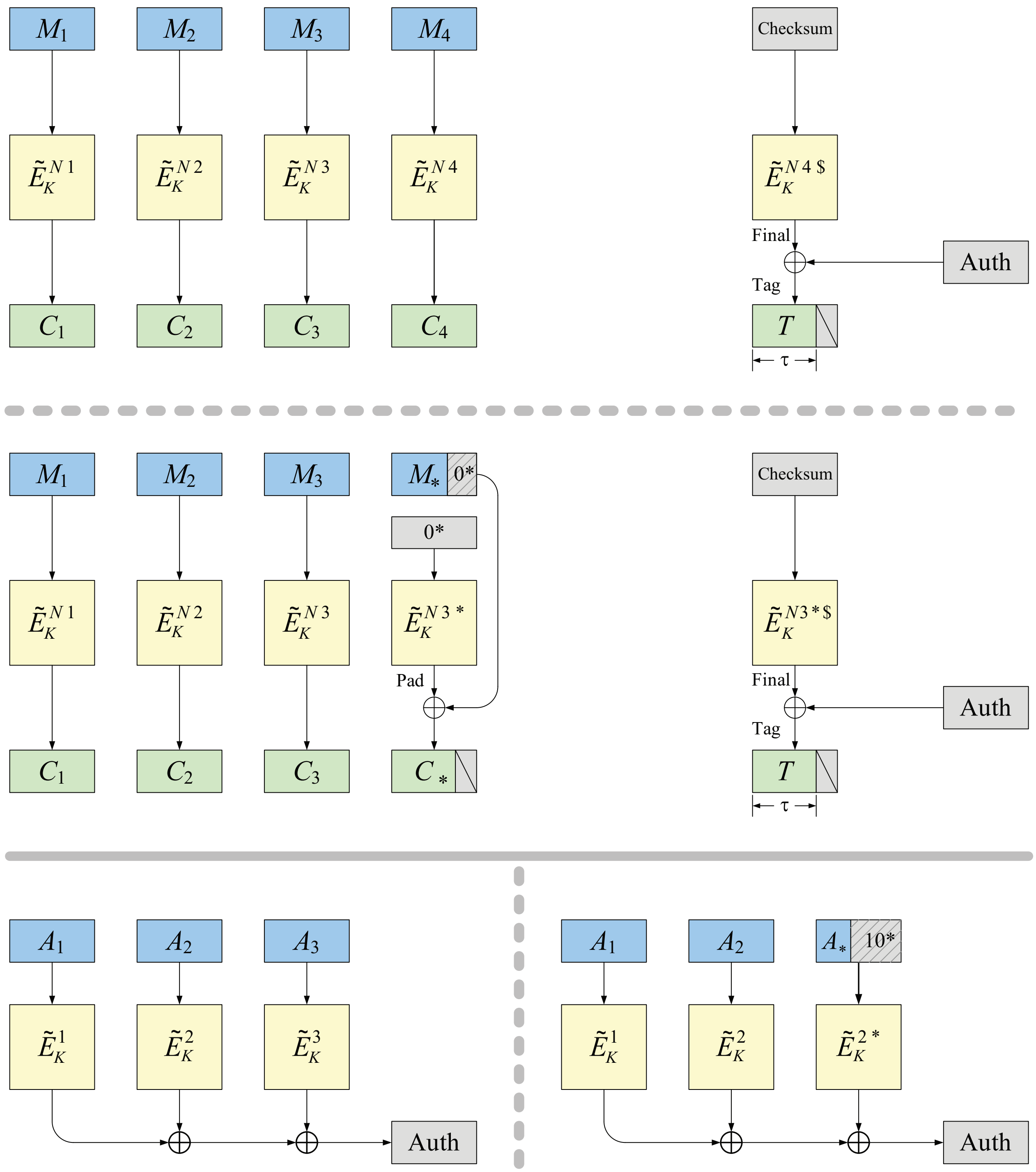

Figure 9: **Illustration of ΘCB3**. The scheme depends on tweakable blockcipher $\widetilde{E}\colon \mathcal{N}\times\mathcal{T}\times\{0,1\}^n \to \{0,1\}^n$ and tag length $\tau \in [0\,..\,n]$. The top figure shows the treatment of a message M having a full final block ($|M_4| = n$) (Checksum $= M_1 \oplus M_2 \oplus M_3 \oplus M_4$) while the middle picture shows the treatment of a message M having a short final block ($1 \le |M_*| < n$) (Checksum $= M_1 \oplus M_2 \oplus M_3 \oplus M_* 10^*$). The bottom-left picture shows the processing of a three-block AD; on bottom-right, an AD with two full blocks and a short one. Algorithm OCB3$[E, \tau]$ coincides with ΘCB3$[\widetilde{E}, \tau]$ for a particular TBC $\widetilde{E} = \mathrm{Tw}[E]$ constructed from E.

$$
\begin{aligned}
\widetilde{E}_K^{N\,i}\,(X) &= E_K(X\oplus\Delta)\oplus\Delta && \text{with } \Delta = \text{Initial} \oplus \lambda_i L && \text{for } i\ge 1\\
\widetilde{E}_K^{N\,i\,*}(X) &= E_K(X\oplus\Delta) && \text{with } \Delta = \text{Initial} \oplus \lambda_i^* L && \text{for } i\ge 0\\
\widetilde{E}_K^{N\,i\,\$}(X) &= E_K(X\oplus\Delta) && \text{with } \Delta = \text{Initial} \oplus \lambda_i^\$ L && \text{for } i\ge 0\\
\widetilde{E}_K^{N\,i\,*\,\$}(X) &= E_K(X\oplus\Delta) && \text{with } \Delta = \text{Initial} \oplus \lambda_i^{*\$} L && \text{for } i\ge 0\\
\widetilde{E}_K^{i}(X) &= E_K(X\oplus\Delta) && \text{with } \Delta = \lambda_i L && \text{for } i\ge 1\\
\widetilde{E}_K^{i*}(X) &= E_K(X\oplus\Delta) && \text{with } \Delta = \lambda_i^* L && \text{for } i\ge 0
\end{aligned}
$$

where

$$
\begin{aligned}
\text{Nonce} &= 0^{127-|N|}\,1\,N & L &= E_K(0^{128})\\
\text{Top} &= \text{Nonce}\wedge 1^{122}\,0^6 & \lambda_i &= 4\,a(i)\\
\text{Bottom} &= \text{Nonce}\wedge 0^{122}\,1^6 & \lambda_i^* &= 4\,a(i)+1\\
\text{Ktop} &= E_K(\text{Top}) & \lambda_i^\$ &= 4\,a(i)+2\\
\text{Stretch} &= \text{Ktop}\parallel \big(\text{Ktop}\oplus(\text{Ktop}\ll 8)\big) & \lambda_i^{*\$} &= 4\,a(i)+3\\
\text{Initial} &= (\text{Stretch}\ll\text{Bottom})[1..128] & a(0) &= 0 \quad \text{//Grey code seq } 0,1,3,2,6,7,5,4,12,\ldots\\
& & a(i) &= a(i-1)\oplus 2^{\text{ntz}(i)} \text{ if } i\ge 1
\end{aligned}
$$

Figure 10: **Definition of** $\widetilde{E} = \text{Tw}[E]$, the tweakable blockcipher built from E.

4.3 Instantiating the TBC

Continuing to assume that $n = 128$ and $\mathcal{N} = \{0,1\}^{<n}$, map each blockcipher $E\colon \mathcal{K}\times\{0,1\}^n \to \{0,1\}^n$ to the TBC $\widetilde{E} = \text{Tw}[E]$, $\widetilde{E}\colon \mathcal{K}\times\mathcal{T}\times\{0,1\}^n\to\{0,1\}^n$, where $\mathcal{T} = \mathcal{N}\times\mathbb{N}_1 \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{*\} \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{\$\} \;\cup\; \mathcal{N}\times\mathbb{N}_0\times\{*\$\} \;\cup\; \mathbb{N}_1 \;\cup\; \mathbb{N}_0\times\{*\}$ by the construction of Figure 10. There, multiplication is in GF(2^{128}) using the irreducible polynomial $\mathtt{x}^{128}+\mathtt{x}^7+\mathtt{x}^7+\mathtt{x}^2+\mathtt{x}+1$. We use the standard facts on the Gray code sequence $a\colon \mathbb{N}_0\to\mathbb{N}_0$ that it is a permutation and $0\le a(i)\le 2i$. It follows that coefficients $\Lambda = \{\lambda_i,\ \lambda_j^*,\ \lambda_j^\$,\ \lambda_j^{*\$}\colon\ 1\le i\le 2^{120},\ 0\le j\le 2^{120}\}$ are distinct and nonzero points of GF(2^{128}). The reader can check that OCB3$[E,\tau]$ = ΘCB3$[\text{Tw}[E],\tau]$.

Security of the constructed TBC. We show that $\widetilde{E} = \text{Tw}[E]$ is a good TBC if E is a good blockcipher. In formalizing this, forward queries may be asked throughout $\mathcal{T}$, but backwards queries must be of the form $\widetilde{E}_K^{N\,i}$.

Lemma 3 *Let $n = 128$ and let $E = \text{Bloc}[n]$ be the ideal blockcipher on n bits. Let $\widetilde{E} = \text{Tw}[E]$, the tweak space being $\mathcal{T}$, and let $\mathcal{T}^{\pm} = \mathcal{N}\times\mathbb{N}_1$. Let $\mathcal{A}$ be an adversary that asks at most q queries, non employing an i-value in excess of 2^{120}. Then $\mathbf{Adv}_{\widetilde{E}}^{\text{prp}[\mathcal{T}^{\pm}]}(\mathcal{A}) \le 6q^2/2^n$.* □

The proof of the lemma is in Appendix A.2. Combining it with Lemma 2 gives Theorem 1.

Acknowledgments

Phil Rogaway had interesting discussions with Tariq Ahmad (University of Massachusetts) on hardware aspects of GCM and OCB3.

The authors appreciate the support of NSF CNS 0904380.

References

[1] M. Bellare and C. Namprempre. Authenticated encryption: relations among notions and analysis of the generic composition paradigm. *J. Cryptology*, 21(4), pp. 469–491, 2008. Earlier version in *ASIACRYPT 2000*.

[2] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. *FSE 2004*, LNCS vol. 3017, Springer, pp. 389–407, 2004.

[3] D. Bernstein. The Poly1305-AES message-authentication code. *FSE 2005*, LNCS vol. 3557, Springer, pp. 32–49, 2005.

[4] D. Bernstein and P. Schwabe. New AES speed records. *INDOCRYPT 2008*, LNCS vol. 5365, Springer, pp. 322-336, 2008.

[5] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, P. Rogaway. UMAC: fast and secure message authentication. *CRYPTO 1999*, LNCS vol. 1666, Springer, pp. 216–233, 1999.

[6] D. Chakraborty and P. Sarkar. A general construction of tweakable block ciphers and different modes of operations. *IEEE Trans. on Information Theory*, 54(5), May 2008.

[7] M. Dworkin. Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality. NIST Special Publication 800-38C. May 2004.

[8] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. November 2007.

[9] P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. *SAC 2002*, LNCS vol. 2595, Springer, pp. 47–61, 2002.

[10] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno. Helix: fast encryption and authentication in a single cryptographic primitive. *FSE 2003*, LNCS vol. 2887, Springer, pp. 330–346, 2003.

[11] V. Gligor and P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. *FSE 2001*, LNCS vol. 2355, Springer, pp. 92–108, 2001.

[12] S. Gueron. Intel's New AES instructions for enhanced performance and security. *FSE 2009*, LNCS vol. 5665, Springer, pp. 51–66, 2009.

[13] S. Gueron and M. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode (revision 2). White paper, available from www.intel.com. May 2010.

[14] S. Halevi. An observation regarding Jutla's modes of operation. Cryptology ePrint report 2001/015. April 2, 2001.

[15] IEEE Standard 802.11i-2004. Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Medium Access Control (MAC) Security Enhancements. 2004.

[16] ISO/IEC 19772. Information technology–Security techniques–Authenticated encryption. First edition, 2009-02-15.

[17] T. Iwata. Authenticated encryption mode for beyond the birthday bound security. *AFRICACRYPT 2008*, LNCS vol. 5023, Springer, pp. 125–142, 2008.

[18] T. Iwata. New blockcipher modes of operation with beyond the birthday bound security. *FSE 2006*, LNCS 4047, pp. 310–327, 2006.

[19] T. Iwata and K. Yasuda. BTM: a single-key, inverse-cipher-free mode for deterministic authenticated encryption. *SAC 2009*, LNCS vol. 5667, Springer, pp. 313–330, 2009.

[20] T. Iwata and K. Yasuda. HBS: a single-key mode of operation for deterministic authenticated encryption. *FSE 2009*, LNCS vol. 5665, Springer, pp. 394–415, 2009.

[21] C. Jutla, Encryption modes with almost free message integrity. *EUROCRYPT 2001*, LNCS vol. 2045, Springer, pp. 529–544, 2001.

[22] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. *CHES 2009*, LNCS 5757, Springer, pp. 1–17, 2009.

[23] J. Katz and M. Yung. Unforgeable encryption and adaptively secure modes of operation. *FSE 2000*, LNCS vol. 1978, Springer, 2001.

[24] T. Kohno, J. Viega, and D. Whiting. CWC: a high-performance conventional authenticated encryption mode. *FSE 2004*, LNCS vol. 3017, Springer, pp. 408–426, 2004.

[25] T. Krovetz. Message authentication on 64-bit architectures. *SAC 2006*, LNCS vol. 4356,

Springer, pp. 327–341, 2006.
[26] C. Leiserson, H. Prokop, and K. Randall. Using de Bruijn sequences to index a 1 in a computer word. Unpublished manuscript. July 7, 1998.
[27] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications* (Revised Edition). Cambridge University Press, 1994.
[28] M. Liskov, R. Rivest, and D. Wagner. Tweakable block ciphers. *CRYPTO 2002*, LNCS vol. 2442, Springer, pp. 31–46, 2002.
[29] S. Lucks. Two-pass authenticated encryption faster than generic composition. *FSE 2005*, LNCS vol. 3557, Springer, pp. 284–298, 2005.
[30] D. McGrew. An interface and algorithms for authenticated encryption. IETF RFC 5116. January 2008.
[31] D. McGrew and J. Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. *INDOCRYPT 2004*, LNCS vol. 3348, Springer, pp. 343–355, 2004. Also Cryptology ePrint report 2004/193, with somewhat different performance results.
[32] OpenSSL: The Open Source Toolkit for SSL/TLS. `http://www.openssl.org/`.
[33] P. Rogaway. Authenticated-encryption with associated-data. *CCS 2002*, ACM Press, 2002.
[34] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. *ASIACRYPT 2004*, LNCS vol. 3329, Springer, pp. 16–31, 2004.
[35] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. on Information and System Security*, 6(3), pp. 365–403, 2003. Earlier version, with T. Krovetz, in *CCS 2001*.
[36] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. *EUROCRYPT 2006*, LNCS vol. 4004, Springer, pp. 373–390, 2006.
[37] B. Tsaban and U. Vishne. Efficient linear feedback shift registers with maximal period. *Finite Fields and Their Applications*, 8(2), pp. 256–267, 2002. Also CoRR cs.CR/0304010, 2003.
[38] VIA Technologies. VIA Padlock programming guide. 2005.
[39] D. Whiting, R. Housley, N. Ferguson. AES encryption & authentication using CTR mode & CBC-MAC. IEEE P802.11 doc 02/001r2, May 2002.
[40] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). IETF RFC 3610. September 2003.
[41] G. Zeng, W. Han, and K. He. High efficiency feedback shift register: σ-LFSR. Cryptology ePrint report 2007/114, 2007.

A Postponed Proofs

A.1 Proof of Lemma 2

Consulting Figure 9 may help. Keep in mind that each $\widetilde{E}_K^{N\,j}$, $\widetilde{E}_K^{N\,j\,\$}$, $\widetilde{E}_K^{N\,j\,*}$, $\widetilde{E}_K^{N\,j\,*\,\$}$, $\widetilde{E}_K^{j}$ and $\widetilde{E}_K^{j\,*}$ is a random permutation on n bits. These permutations are all independent. To emphasize that these are random permutations we write π in place of $\widetilde{E}_K$.

Privacy. During the adversary's attack it asks queries $(N^1, A^1, M^1), \ldots, (N^q, A^q, M^q)$. Since the N^i-values must be distinct each permutation $\pi^{N^i\,\cdots}$ is used at most once. We are thus applying independent random permutations to a single point, so all of the outputs are uniformly random and independent. Now in three of the four cases—$\pi^{N\,j\,\$}$, $\pi^{N\,j\,*}$, and $\pi^{N\,j\,*\,\$}$—the permutation's output is not returned to the adversary, but is, instead, either xored with Auth^i (then truncated to τ bits) or xored it with $M^i_*\,0^*$ (then truncated to $|M^i_*|$ bits). Either way, the result remains uniform and independent of all other outputs, as $M^i + *$, Auth^i, and τ are independent of the $\pi^{N^i\cdots}$

values. We conclude that the result from the adversary's i^{th} query is a uniformly random string of length $|M^i|+\tau$, independent of all other query responses. This implies that the adversary's privacy advantage is zero.

AUTHENTICITY. Before we launch into proving authenticity, consider the following simple game, which we call game G. Suppose that you know that an n-bit string X is *not* some particular value X_0. All of the 2^n-1 other values are equally likely. Then your chance of correctly predicting the τ-bit prefix of X is at most $2^{n-\tau}/(2^n-1)$. That's because the best strategy is to guess any τ-bit string other than the τ-bit prefix of X_0. The probability of being right under this strategy is $2^{n-\tau}/(2^n-1)$. We will use this fact in the sequel.

Now suppose that the adversary asks a sequence of queries $(N^1, A^1, M^1), \ldots, (N^q, A^q, M^q)$ and then makes its forgery attempt $(N, A, \mathcal{C})$. Let $M^i = M^i_1 \cdots M^i_{m_i}$ or $M^i = M^i_1 \cdots M^i_{m_i} M^i_*$ be the message queries, let $A^i = A^i_1 \cdots A^i_{a_i}$ or $A^i = A^i_1 \cdots A^i_{a_i} A^i_*$ be the AD queries, and so on, superscripts being used to indicate the query number. Let $(N, A, \mathcal{C})$ be the forgery attempt, $\mathcal{C} = C \,\|\, T$, $C = C_1 \cdots C_c$ or $C = C_1 \cdots C_c C_*$, and so on, absent superscripts indicating that the quantity in question belongs to the attempted forgery. We distinguish the following cases for the forgery attempt:

(1) Suppose $N \notin \{N^1, \ldots, N^q\}$. Then the adversary needs to find the correct value of $T = \pi^{N\cdots}(\text{Checksum}) \oplus \text{Auth}$ but has seen no image $\pi^{N\cdots}(\,)$. The chance that the adversary can do this is clearly $2^{-\tau}$.

(2) Suppose $N = N^i$ and one of C^i and C has length divisible by n, but the other does not. We may ignore queries other than the i^{th} since the responses to such queries are unrelated to the adversary's task of producing a valid ciphertext $(N, A, \mathcal{C})$ with $N = N^i$. We may ignore the values Auth_i and Auth, even allowing the adversary to know or to select these strings. As with (1), the adversary needs to find the correct value of T but has seen no image for the relevant random permutation: it needs to guess an $\text{Auth} \oplus \pi^{N\,c\,*\,\$}(\,)$ value but no $\pi^{N\,c\,*\,\$}$ has been used; or else it needs to guess an $\text{Auth} \oplus \pi^{N\,c\,\$}(\,)$ value but no $\pi^{N\,c\,\$}$ has been used. The chance that the adversary can forge in this case is at most $2^{-\tau}$.

(3) Suppose $N = N^i$, $m_i \neq c$, and either C^i and C both have length divisible by n or neither C^i nor C have length divisible by n. This is like case (2).

(4) Suppose $N = N^i$, $A^i \neq A$. We may ignore queries other than the i^{th} since the responses to such queries are unrelated to the adversary's task at hand. Having made this simplification—the adversary asks a single (N^i, A^i, M^i) query and must then forge using the same nonce but a different AD—suppose that we provide the adversary with all the $\pi^{N\cdots}$ permutations. Even then the adversary will have small chance to produce a valid forgery. To forge in this setting the adversary's job amounts to guessing the first τ bits of Auth. The only relevant information it has for doing this is the first τ bits of Auth^i. Suppose we give the adversary all of Auth^i instead. A case analysis is needed. If the adversary selects A but not A^i to have a multiple of n bits, or it selects A^i but not A to have a multiple of n bits, then its chance to guess the first τ bits of Auth will be $2^{-\tau}$. Otherwise, if the adversary selects $a \neq a_i$ then its chance to guess the first τ bits of Auth will again be $2^{-\tau}$. Otherwise we are in the setting where $a = a_i$ and both A and A^i are multiples or else non-multiples of n bits. The two cases are similar; assume the former. Since $A^i \neq A$ we know that they differ on some particular block, say $A^i_j \neq A_j$. Then even if we give the adversary π^k for all $k \neq j$, and give it $\pi^j(A^i_j)$ as well, still the adversary will not be able to do well at guessing $\pi^j(A_j)$, and therefore it will not be able to do well at guessing $\text{Auth} = \oplus_{j=1}^{a} \pi^j(A_j)$. Namely, we are now in the setting of game G, and the adversary's chance to succeed is $2^{n-\tau}/(2^n-1)$.

(5) Suppose $N = N^i$, $A = A^i$, $m = c$, and $|M^i| = |C^i| = |C|$ are divisible by n. We may again ignore the queries other than the i^{th}. For simplicity, imagine that we reveal to the adversary π^j and $\pi^{j\,*}$, so Auth and Auth_i are adversarially known. If the forgery attempt has the form $(N, A, C^i \,\|\, T^i)$ where $C^i = C$ then the adversary's chance of forging is zero: the adversary is not allowed to repeat a known $(N^i, A^i, \mathcal{C}^i)$ verbatim, and changing T from T^i will make this forgery attempt incorrect. We may therefore assume that the forgery attempt is $(N, A, C^i \,\|\, T^i)$ where $C^i \neq C$, say $C^i_j \neq C_j$ for some particular $j \in [1 \mathinner{..} m]$. The changed C_j makes the corresponding M_j is almost unpredictable; all that is known of it is that it is not M^i_j, but all remaining $2^n - 1$ possibilities are equally likely. Even if we provide the adversary all $M_1, \ldots, M_m$ except M_j, this means that the Checksum will be any of $2^n - 1$ values, all equally likely. Even if we make $\pi^{N\,m\,\$}$ public, its output, Final, will be any of $2^n - 1$ values, all equally likely. We are again in the setting of game G, and the adversary's chance to win is $2^{n-\tau}/(2^n - 1)$.

(6) Suppose $N = N^i$, $A = A^i$, $m = c$, and $|M^i| = |C^i| = |C|$ are not divisible by n. We proceed as in case (5), but must distinguish the following: $C^i_j \neq C_j$ for some $j \in [1 \mathinner{..} m]$, or else $C^i_j = C_j$ for all $j \in [1 \mathinner{..} m]$, but $C^i_* \neq C_*$. The first of these subcases proceeds as with case (5), so assume the second. We may this time provide the adversary all $\pi^{N\,i}$ and $\pi^{N\,i\,*}$ values, so that the adversary will in fact know Checksum_i and Checksum, which are necessarily distinct. (Here it is important that we used 10*-padding for in the contribution of M_* to the Checksum). The adversary can be assumed to know all of Final^i, but still its chance to predict the image of Checksum will be at most $2^n - 1$, and, by game G, its ability to predict the first τ bits of Final, and thus T is $2^{n-\tau}/(2^n - 1)$.

A.2 Proof of Lemma 3

We generalize the adversary's capabilities in attacking $\text{Tw}[E]$; see Figure 11 for the construction we'll call TW. There we write π instead $\widetilde{E}_K$. The adversary, which we still refer to as $\mathcal{A}$, may now ask queries we'll refer to as being of TYPE-1, TYPE-2, TYPE-3a, TYPE-3b. In other words, the adversary's queries may take any of the forms $(1, W, \lambda)$, $(2, W, \text{Top}, \text{Bottom}, \lambda)$, $(3\text{a}, W, \text{Top}, \text{Bottom}, \lambda)$, or $(3\text{b}, Z, \text{Top}, \text{Bottom}, \lambda)$. We insist that the adversary not ask a query with $\text{Top} = 0$ (we stop to distinguish field points and the corresponding strings) and we demand that any $\lambda \in \text{GF}(2^n)$ asked in a query is used only for queries of one numeric TYPE (it's fine to use the same λ in queries of TYPE-3a and 3b). The adversary may not repeat queries nor ask a query with a trivially known answer (a TYPE-3b query following the corresponding TYPE-3a query, or the other way around). Working in $\text{GF}(2^n)$, we sometimes write xor as addition.

As the adversary asks its queries the mechanism makes what we will call *internal* queries to the random-permutation π. For example, the adversary's TYPE-1 query of (W, λ) results in an internal type-1 query of X. The internal queries come in two flavors, *direct* and *indirect*, as show in Figure 11. Note that the total number of internal queries resulting from the adversary's q queries is at most $\sigma = 2q+1$. The hash function H that we use to compute Initial is the map defined and and proven secure in Lemma 1. That said, any strongly xor-universal hash function with the needed domain and range will do. It is important to understand that all of the abilities present in a "real" adversary attacking $\text{Tw}[E]$ are also represented in the abilities of an adversary attacking $\text{TW}[E]$ we have now described.

We aim to show $\mathcal{A}$ will get small advantage in attacking TW. The proof involves a game-playing argument followed by a case analysis of some collision probabilities. We begin with a game 1 that perfectly simulates the TW-construction. As the adversary $\mathcal{A}$ asks queries the game grows the permutation π in the usual way, preparing each input for π or π^{-1} exactly as would TW. The

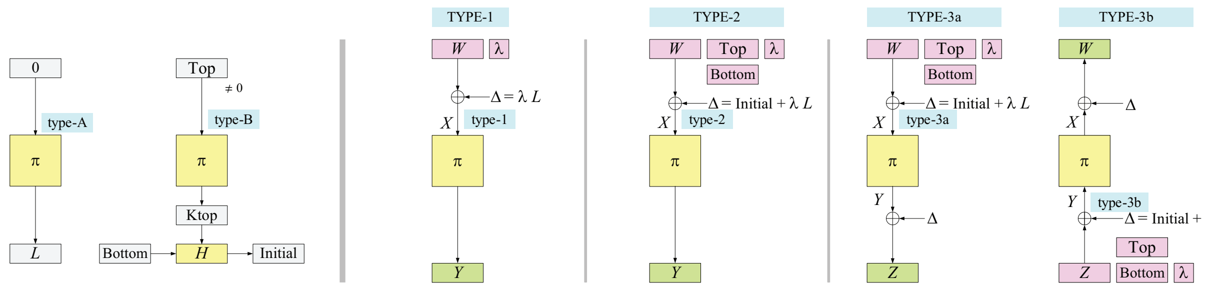

Figure 11: **The TW construction**. The adversary's queries (TYPE 1, 2, 3a, 3b) result in *internal* queries that are either *direct* (type 1, 2, 3a, 3b) or *indirect* (type A, B). The proof of Lemma 3 hinges on establishing the rarity of nontrivial collisions among the inputs or outputs of π.

responses to type-A and B queries are stored and looked up as needed. In game 2 we return, in response to each internal query π or π^{-1}, a freshly minted uniformly random point of $\mathrm{GF}(2^n)$. Note that this results in values returned to the adversary that are, likewise, uniformly random. In game 3 we perfectly simulate an ideal tweakable blockcipher $\widetilde{\pi}$ (with the right domain and tweak space). By the "switching lemma" the advantage of $\mathcal{A}$ in distinguishing games 2 and 3 is at most $0.5\,q(q-1)/2^n$, so we must only bound the gap between games 1 and 2.

In game 1, consider answering each internal query by uniformly sampling from $\{0,1\}^n$ and, hopefully, returning that sample. If we have already used our speculatively generated return value set a flag *bad* and re-sample from the co-range (for π-queries) or co-domain (for π^{-1} queries). The above *bad*-setting events occur with probability at most $0.5\,\sigma(\sigma-1)/2^n$.

When an internal query clashes with any prior commitment made then, to accurately play game 1, we must answer the query according to the prior commitment. Assume we do so, and then set *bad*. Call these *bad*-setting events *collisions*. We can write games 1 and 2 so as to be identical until *bad* is set, so we have only to bound the probability of collisions in game 2, the version where we uniformly sample for responding to internal queries. Note that game 2 maintains the invariant that values returned to the adversary are independent of the values L and Initial selected internally. Because of this, we can simplify the temporal aspect of the game and replace it by an alternative one in which the adversary chooses all TBC queries, *and their responses*, at the beginning. Then we make the indirect queries that determine L and Initial, and determine if a collision has occurred. Excising adaptivity in such a manner has been illustrated in much prior work.

Any potential collision event—eg, the 20^{th} internal query colliding with the 6^{th}—can be summarized by writing something like $\mathsf{Coll}(3\mathrm{a}, 1)$, interpreted as saying that first there was a type-3a internal query $(W, \mathrm{Top}, \mathrm{Bottom}, \lambda)$, which generated a π-input of X (its value to be determined) and an adversarially-known response Z, and then the adversary asked a type-1 query of (W', λ'), which gave rise to a π-input of X' (value to be determined), and an adversarially-known response of Y'. Now we make the underlying type-A and type-B queries and it so happens that $X = X'$. Such an event is unlikely since it implies that $W + \mathrm{Initial} + \lambda L = W' + \lambda' L$, and Initial is uniform and independent of all other values named in the formula: we select the type-B output Ktop of π uniformly at random, and H is universal-1, making $H_{\mathrm{Ktop}}(\mathrm{Bottom})$ uniform, too. The probability of the event happening, for a given pair of indirect queries, is at most 2^{-n}. The same holds for each of the 36 possible collision types. To avoid tedious repetition, we provide a few examples. We continue to use the same convention as in the last example, priming variables for the second query.

- $\Pr[\mathsf{Coll}(\mathrm{A}, \mathrm{B})] = \Pr[\mathsf{Coll}(\mathrm{B}, \mathrm{A})] = 0$ since the adversary is not allowed to query with $\mathrm{Top} = 0$.
- $\Pr[\mathsf{Coll}(3\mathrm{a}, 3\mathrm{a})] = \Pr[W + \mathrm{Initial} + \lambda L = W' + \mathrm{Initial}' + \lambda' L]$. Queries may not be repeated, so $(W, \mathrm{Top}, \mathrm{Bottom}, \lambda) \neq (W', \mathrm{Top}', \mathrm{Bottom}, \lambda')$. Suppose $\mathrm{Top} \neq \mathrm{Top}'$. Then Ktop and Ktop' are uniform and independent, making Initial and $\mathrm{Initial}'$ uniform and independent

Definition of $S_i(X)$		32-Bits					64-Bits		128-Bits	
		x86	arm	ppc	sun	mips	x86	sun	x86	ppc
S_1	$(X\ll 1) \oplus (\mathrm{msb}(X) \cdot 135)$	10.9	22.9	13.4	23.7	18.6	4.0	3.7	6.8	5.0
S_2	$(B, (A\ll 1) \oplus (\mathrm{msb}(A) \cdot 10^{120}1010001) \oplus B)$	7.6	17.0	5.2	13.2	12.6	3.3	2.6	7.1	6.0
S_3	$(B,\ (A\ll 1) \oplus (A\gg 1) \oplus (B \wedge 148))$	7.7	11.4	4.3	13.3	14.2	4.1	2.7	5.5	7.1
S_4	$(C,\ D,\ B,\ (A\ll 1) \oplus (\mathrm{msb}(A) \cdot 831) \oplus B \oplus D)$	3.0	6.2	2.5	4.5	6.6	5.3	5.3	9.4	8.1
S_5	$(C,\ D,\ B,\ (A\ll 1) \oplus (A\gg 1) \oplus (D \wedge 107))$	4.1	4.4	2.1	4.2	5.8	5.7	5.2	5.5	6.0
S_6	$(C,\ D,\ B,\ (A\ll 1) \oplus (A\gg 1) \oplus (D\ll 15))$	4.0	3.4	2.0	4.2	5.7	5.6	5.3	8.2	8.1

Figure 12: **Some maximal-length 128-bit LFSRs and their performance.** The input $X \in \{0,1\}^{128}$ is partitioned into $X = A \,\|\, B$ (for S_2 or S_3) or $X = A \,\|\, B \,\|\, C \,\|\, D$ (for S_4, S_5, or S_6). Repeated application of $S_i \colon \{0,1\}^{128} \to \{0,1\}^{128}$ to any $X \in \{0,1\}^{128} \setminus \{0^{128}\}$ yields all strings in $\{0,1\}^{128} \setminus \{0^{128}\}$. The table gives the time to compute S_i, in CPU cycles, averaged over a large number of runs, when inputs and outputs are provided to the implementation using registers of 32, 64, and 128 bits.

by the universal-1 property of H, so the probability in question is at most 2^{-n}. Suppose Top $=$ Top$'$ but Bottom $\neq$ Bottom$'$. Then Ktop $=$ Ktop$'$ is uniform and, by the xor-universality of H, variables Initial and Initial$'$ are uniform and independent of each other and of every other variables appearing in the formula, making the probability in question at most 2^{-n}. If Top $=$ Top$'$ and Bottom $=$ Bottom$'$ and $\lambda = \lambda'$ then we know that $W \neq W'$ and the probability of collision is 0. Finally, if Top $=$ Top$'$ and Bottom $=$ Bottom$'$ and $W = W'$ then we know that $\lambda \neq \lambda'$ and the probability we are considering collapses to $\Pr[\lambda L = \lambda' L] = \Pr[cL = 0]$ where $c = \lambda - \lambda' \neq 0$. Since L was chosen uniformly at random in the type-A query, only the choice of $L = 0$ results in a collision, which happens with probability 2^{-n}.

- $\Pr[\mathsf{Coll}(2, 3\mathrm{b})] = \Pr[Y = \mathrm{Initial}' + \lambda' L]$. This is at most 2^{-n} as, for example, Initial$'$ is uniform and independent of Y, Λ', and L.
- $\Pr[\mathsf{Coll}(3\mathrm{b}, 2)] = \Pr[W + \mathrm{Initial} + \lambda L = W' + \mathrm{Initial}' + \lambda' L$. Since λ-values must be distinct between TYPE-2 and TYPE-3b the probability is $\Pr[cL = W + \mathrm{Initial} + W' + \mathrm{Initial}']$ for some $c \neq 0$. Since the RHS side is independent of L and L is uniform, the probability is at most 2^{-n}.

Continuing in this way one finds that each type of collision occurs with probability at most 2^{-n}, implying a probability for any collision of at most $0.5\,\sigma(\sigma-1)/2^n$. Summing with the addends of $0.5\,\sigma(\sigma-1)$ and $0.5\,q(q-1)$ and recalling that $\sigma \leq 2q+1$ we conclude that the total adversarial advantage is at most $4.5q^2 + 1.5q \leq 6q^2$, completing the proof.

B New Word-Oriented LFSRs

Recall that in OCB2 each 128-bit offset is computed from the prior one by multiplying it, in $\mathrm{GF}(2^{128})$, by the constant $\mathtt{x} = 2 = 0^{126}10$. Concretely, the point $X \in \{0,1\}^{128}$ is stepped (or "incremented" or "doubled") by applying the map $S_1(X) = (X\ll 1) \oplus (\mathrm{msb}(X) \cdot 135)$. The constant 135 (decimal) represents (without the $\mathtt{x}^{128}$ term) the primitive polynomial $g(\mathtt{x}) = \mathtt{x}^{128} + \mathtt{x}^7 + \mathtt{x}^2 + \mathtt{x} + 1$.

Chakraborty and Sarkar suggested [6] that there might be an incrementing function more efficient than S_1; they suspected that one might achieve efficiency gains with a *word-oriented* LFSR [37], as exemplified by the blockcipher SNOW [9]. After all, multiplication by $\mathtt{x}$ and reducing mod $g(\mathtt{x})$ is just the "Galois configuration" of a particular 128-bit LFSR [27], and one that has not been optimized for software performance. Some other 128-bit LFSRs might run faster.

To develop this idea, let S be an $n \times n$ binary matrix that is invertible over GF(2). Then we may regard S as the feedback matrix of an LFSR that transforms the row vector $X \in \{0,1\}^n$ into the row vector $X \cdot \mathsf{S}$, a process we refer to as *stepping* the string X under S. The t-fold stepping of X by S is realized by matrix S^t. If the characteristic polynomial of S is primitive (over GF(2)) then the order of S in the general linear group $GL(n, \mathrm{GF}(2))$ will be $2^n - 1$ and the map $X \mapsto X \cdot \mathsf{S}$ will have two cycles: the length-1 cycle from 0^n to itself and the cycle of length $2^n - 1$ passing through all remaining n-bit strings [27]. The matrices $\langle \mathsf{S} \rangle = \{\mathsf{S}^i : 1 \le i \le 2^{n-1} - 1\}$, along with the matrix $n \times n$ zero matrix, can be regarded as a representation of $\mathrm{GF}(2^n)$ under the operations of matrix multiplication and matrix addition, both mod 2.

Based on the paragraph above, the following is a simple way to obtain maximal and fast-to-compute 128-bit LFSRs. Generate candidate LFSRs by randomly combining a small number of shifts, ands, xors, using small or random constants. Represent each scheme by its feedback matrix. For each candidate matrix, check if it has a primitive characteristic polynomial. This is roughly the same approach taken by Zeng, Han, and He [41] to devise some software-efficient maximal-period shift registers intended for stream-cipher use. Using it, we generated and tested thousands of 128-bit stepping functions. Some efficient-to-compute schemes giving rise to maximal LFSRs are shown in Figure 12. Our experience searching for such maximal LFSRs suggests that they are rather finicky and sparse.

Implementing the candidate LFSRs on a variety of platforms revealed no clear winner; see Figure 12. Beyond this, we found that none of the stepping functions were competitive with xoring in a pre-computed 128-bit value. All of the candidate stepping function introduce endian favoritism. In the end, then, we decided against using an LFSR stepping function to update offsets, going back to the OCB1 approach, instead.