The Design and Evolution of OCB

Ted Krovetz

Computer Science Department, California State University, 6000 J Street,
Sacramento, California 95819, USA
tdk@csus.edu

Phillip Rogaway

Department of Computer Science, Kemper Hall of Engineering, University of California,
Davis, California 95616, USA
rogaway@cs.ucdavis.edu

Communicated by Tetsu Iwata.

Received 20 December 2019 / Revised 26 August 2020 / Accepted 14 September 2020
Online publication 27 July 2021

Abstract. We describe OCB3, the final version of OCB, a blockcipher mode for authenticated encryption (AE). We prove the construction secure, up to the birthday bound, assuming its underlying blockcipher is secure as a strong-PRP. We study the scheme's software performance, comparing its speed, on multiple platforms, to a variety of other AE schemes. We reflect on the history and development of the mode.

Keywords. AEAD, Authenticated encryption, CAESAR competition, Cryptographic standards, Fast software encryption, Modes of operation, OCB.

1. Introduction

Schemes for authenticated encryption (AE) symmetrically encrypt a message in a way that ensures both its confidentiality and authenticity. OCB is a well-known algorithm for achieving this aim. It is a blockcipher mode of operation, the blockcipher usually being AES.

There are three main variants of OCB. The first, now called OCB1 (2001) [39], was motivated by Charanjit Jutla's IAPM [24]. A second version, now called OCB2 (2004) [18, 38], added support for *associated data* (AD) [37] and redeveloped the mode using the idea of a tweakable blockcipher [30]. OCB2 was recently found to have a disastrous bug [17]. The final version of OCB, called OCB3 (2011) [26], corrected some missteps taken with OCB2 and achieved the best performance yet. It is specified in RFC 7253 [27] and was selected for the CAESAR final portfolio [7].

This paper is about OCB3: its definition, development, security, and software performance. We update the proceedings paper on OCB3 [26], freshening all of the experimental results, expanding the proof, and placing the entire enterprise in context. When we speak of OCB in this paper, we will henceforth mean OCB3.

OCB encryption protects both the confidentiality and authenticity of a plaintext M and, additionally, the authenticity of an associated data A and nonce N . The nonce, a string of 120 or fewer bits, must be unique to each encryption call. OCB does its work using a 128-bit blockcipher E . Encryption needs at most $m + a + 2$ blockcipher calls, where $m = \lceil |M|/128 \rceil$ is the block size of the plaintext and $a = \lceil |A|/128 \rceil$ is the block size of the AD. If the nonce is implemented as a counter and the implementation caches a secret 16-byte value with each message encrypted, then $63/64 \approx 98\%$ of the time the number of blockcipher calls can be reduced to $m + a + 1$. If the AD is fixed during a session, then after processing it the first time, there is effectively no computational cost for subsequent authentications of it and the addend of a should be ignored. OCB requires a single key K for the underlying blockcipher, and all blockcipher calls are keyed with it. OCB is online in the sense that one need not know the length of A or M to proceed with encryption, nor need one know the length of A or the ciphertext C to proceed with decryption. OCB is fully parallelizable: almost all of its blockcipher calls can be performed simultaneously. Computational work beyond blockcipher calls is restricted to a small number of logical operations per call.

OCB enjoys provable security: the mode is a secure AE scheme, under the standard definition [37,39], assuming that the underlying blockcipher E is secure as a strong pseudorandom permutation (PRP). As with many modes of operation, security degrades quadratically in the number of blocks processed.

The starting point for all versions of OCB was to insist on a parallelizable design based on a 128-bit blockcipher, presumably AES. We aimed to minimize overhead, understood as any work beyond a single AES-call per input block. As time has gone on, this starting point has only come to seem better. This is because a growing fraction of general-purpose CPUs include hardware support for AES, and the quality of AES acceleration has been steadily improving. On an Intel Cannon Lake processor, we see peak speeds for OCB of about 0.43 cpb (cycles per byte). On an ARM Cortex-A73 processor, we see peak speeds of about 1.14 cpb. These speeds are about 30% and 40% faster, respectively, than what we observe for GCM [10]. Securely encrypting at such speeds on a general-purpose processor might have seemed unrealistic even a decade ago.

We now move on to provide some preliminaries on blockciphers and AE schemes. We then give two equivalent descriptions of OCB—the first based on a conventional blockcipher, like AES, the second based on a tweakable blockcipher [30] constructed from an ordinary blockcipher. Next we describe some of the design choices underlying OCB, trying to sketch how the mechanism came to take its final form. After that, we describe our experimental work, comparing the software performance of OCB and five other AE schemes. Our experiments were carried out on three disparate platforms. Our performance study is limited to software; for hardware studies of CAESAR candidates, including OCB, see the work from Kris Gaj's lab at GMU [8,15]. Then comes a proof of security. We conclude with some comments about OCB and its significance, identifying some of the lessons learned along the way.

2. Preliminaries

We begin with a few preliminaries, defining blockciphers, tweakable blockciphers, and AE schemes, and how we will measure the security of each.

BLOCKCIPHERS. A *blockcipher* is a function $E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ where the *key space* $\mathcal{K}$ is a finite nonempty set, the *block size* $n \geq 1$ is a number, and $E(K, \cdot) = E_K(\cdot)$ is a permutation for each $K \in \mathcal{K}$. Define $E^{-1} : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ from E by saying that $E^{-1}(K, Y) = E_K^{-1}(Y)$ is the unique point X such that $E(K, X) = Y$.

OCB requires a blockcipher E with a block size of $n = 128$ bits. It is possible to define OCB for other block sizes [25], but this paper does not. We usually have in mind that OCB's blockcipher E is AES (which always has a block size of $n = 128$ bits). All of our performance studies assume that E is AES-128, the version of AES with key space $\mathcal{K} = \{0, 1\}^{128}$.

We next define the PRP (pseudorandom permutation) and strong-PRP security for a blockcipher E . With $E : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ a blockcipher and $\mathcal{A}$ an adversary (an algorithm with access to one or more oracles), define the strong-PRP security of E as

$$\text{Adv}_E^{\pm\text{prp}}(\mathcal{A}) = \Pr[\mathcal{A}^{E_K(\cdot), E_K^{-1}(\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot), \pi^{-1}(\cdot)} \Rightarrow 1]$$

where $K \leftarrow \mathcal{K}$ is selected uniformly from $\mathcal{K}$ in the experiment underlying the left-hand addend and $\pi(\cdot) \leftarrow \text{Perm}(n)$ is selected uniformly from $\text{Perm}(n)$, the set of all permutations on $\{0, 1\}^n$, in the experiment underlying the right-hand addend. When we write $\mathcal{A}^{\mathcal{O}_1, \mathcal{O}_2, \dots} \Rightarrow 1$, we mean the event that $\mathcal{A}$ with the specified oracles outputs the bit 1 and halts. Define the (ordinary) PRP-security of E as

$$\text{Adv}_E^{\text{prp}}(\mathcal{A}) = \Pr[\mathcal{A}^{E_K(\cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot)} \Rightarrow 1]$$

by removing the second oracle in each experiment.

The *ideal* blockcipher of block size n is the blockcipher $\mathbb{E}^n : \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ where each key $K \in \mathcal{K}$ names a distinct permutation. In this blockcipher, the key space $\mathcal{K}$ has $2^n!$ keys, each corresponding to a table specifying an n -bit permutation.

TWEAKABLE BLOCKCIPHERS. Liskov, Rivest, and Wagner (LRW) put forward the notion of a *tweakable blockcipher* [30], and they demonstrated that an OCB1-like construction could more easily be designed and proven secure based on this abstraction than a conventional blockcipher. Our work builds on this insight, viewing OCB as based either on a blockcipher or on a tweakable blockcipher (in which case we adjust the name to ΘCB).

A tweakable blockcipher (TBC) is a function $\tilde{E} : \mathcal{K} \times \mathcal{T} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ where the key space $\mathcal{K}$ is a finite nonempty set, or a set that otherwise has an understood distribution on it; the tweak space $\mathcal{T}$ is a nonempty set; and the block size $n \geq 1$ is a number. We insist that $\tilde{E}(K, T, \cdot)$ be a permutation on $\{0, 1\}^n$. Its inverse $\tilde{D} = \tilde{E}^{-1}$ is defined by letting $\tilde{D}(K, T, Y) = X$ when $\tilde{E}(K, T, X) = Y$. We may write $\tilde{E}_K^T(\cdot) = \tilde{E}(K, T, X)$ and $\tilde{D}(K, T, Y) = \tilde{D}_K^T(Y)$.

For a granular notion of security for the TBC $\tilde{E} : \mathcal{K} \times \mathcal{T} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$, we identify each tweak as either a *unidirectional* tweak (only forward queries are allowed

for it) or as a *bidirectional* one. (Both forward and backward queries are allowed.) Intuitively, this distinction is useful because, when making a TBC from a conventional blockcipher, unidirectional tweaks only need tweak-dependent “pre-whitening” but bidirectional tweaks need “post-whitening” too [30,38]. Formally, one names the subset $\mathcal{B} \subseteq \mathcal{T}$ of bidirectional tweaks, understanding that the remaining tweaks $\mathcal{U} = \mathcal{T} \setminus \mathcal{B}$ are unidirectional. We associate with the TBC $\tilde{E}: \mathcal{K} \times \mathcal{T} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ and the subset of tweaks $\mathcal{B} \subseteq \mathcal{T}$ the partitioned-tweakspace PRP measure

$$\mathbf{Adv}_{\tilde{E}}^{\mathcal{B}\text{-prp}}(\mathcal{A}) = \Pr[\mathcal{A}^{\tilde{E}_K(\cdot, \cdot), \tilde{D}_K(\cdot, \cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\pi(\cdot, \cdot), \pi^{-1}(\cdot, \cdot)} \Rightarrow 1]$$

where $K \leftarrow \mathcal{K}$ is chosen uniformly at the beginning of the left-hand experiment, where $\pi \leftarrow \text{Perm}(\mathcal{T}, n)$ is chosen uniformly at the beginning of the right-hand experiment, and where adversary $\mathcal{A}$ may not ask any decryption (second-oracle) query (T, Y) with $T \in \mathcal{T} \setminus \mathcal{B}$. Any such (invalid) query would get empty-string response. By $\text{Perm}(\mathcal{T}, n)$ we mean the space of all permutations $\pi(\mathcal{T}, \cdot)$ on $\{0, 1\}^n$, one for each $T \in \mathcal{T}$.

The partitioned-tweakspace PRP notion above unifies tweakable-PRP and strong-tweakable-PRP security: with $\tilde{E}: \mathcal{K} \times \mathcal{T} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ a TBC we can resurrect the notion for a strong-tweakable-PRP [30] by defining $\mathbf{Adv}_{\tilde{E}}^{\pm\text{prp}}(\mathcal{A}) = \mathbf{Adv}_{\tilde{E}}^{\mathcal{T}\text{-prp}}(\mathcal{A})$. We can realize the conventional (not strong) version [30] by defining $\mathbf{Adv}_{\tilde{E}}^{\text{prp}}(\mathcal{A}) = \mathbf{Adv}_{\tilde{E}}^{\emptyset\text{-prp}}(\mathcal{A})$. Going a step further, the PRP and strong-PRP security notions for a conventional blockcipher $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ can be recovered by regarding the tweak space as a singleton set.

The *ideal* TBC with tweak space $\mathcal{T}$ and block size n is the TBC $\tilde{\mathbb{E}}^{\mathcal{T}, n}: \mathcal{K} \times \mathcal{T} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ where each key $K \in \mathcal{K}$ and $T \in \mathcal{T}$ names a distinct permutation $\mathbb{E}^n(K, T, \cdot)$ on $\{0, 1\}^n$.

AE SCHEMES. Following traditions rooted in prior work [4,28,37,39], we regard an AE scheme (meaning a nonce-based AE scheme with associated-data, sometimes referred to as an AEAD scheme [37]) as a triple $\Pi = (\Pi.\mathcal{K}, \Pi.\mathcal{E}, \Pi.\mathcal{D}) = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The *key space* $\mathcal{K}$ is a finite nonempty set (or a set that otherwise has an understood distribution). The *encryption algorithm* $\mathcal{E}$ takes in a *key* $K \in \mathcal{K}$, a *nonce* $N \in \{0, 1\}^*$, a string of *associated data* $A \in \{0, 1\}^*$, and a *plaintext* $M \in \{0, 1\}^*$. It returns, deterministically, a string $C = \mathcal{E}(K, N, A, M)$ that is either a string $C \in \mathcal{C} = \{0, 1\}^*$, the *ciphertext*, or the symbol $\perp$. The former must happen if and only if $(K, N, A, M) \in \mathcal{K} \times \mathcal{N} \times \mathcal{M} \times \mathcal{A}$ for nonempty sets $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{M}$, the *nonce space*, *AD space*, and *message space* of Π . The *decryption algorithm* $\mathcal{D}$ takes in a tuple (K, N, A, C) and deterministically returns a value $\mathcal{D}(K, N, A, C)$ that is either a string M or the distinguished symbol $\perp$. The latter must happen if $(K, N, A, C) \notin \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C}$. We require that $\mathcal{D}(K, N, A, C) = M$ when $C = \mathcal{E}(K, N, A, M)$ is a string. We require of the message space $\mathcal{M}$ that $M \in \mathcal{M}$ implies $M' \in \mathcal{M}$ when $|M'| = |M|$. We require that $|\mathcal{E}(K, N, A, M)| = |\mathcal{E}(K, N, A, M')| = |M| + \tau$ when the encryptions are strings and $|M| = |M'|$. We call τ the *tag length* or *expansion* of the AE scheme. We may write $\mathcal{E}(K, N, A, M)$ as $\mathcal{E}_K(N, A, M)$ or $\mathcal{E}_K^{N,A}(M)$, and similarly for $\mathcal{D}$.

Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an AE scheme with tag length τ . Given an adversary $\mathcal{A}$, we measure how well it breaks the privacy of (here meaning the confidentiality) of Π by the real number

$$\text{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathcal{E}_K(\cdot, \cdot, \cdot)} \Rightarrow 1] - \Pr[\mathcal{A}^{\$(\cdot, \cdot, \cdot)} \Rightarrow 1] .$$

The first addend is the probability that $\mathcal{A}$ outputs 1 when given the oracle $\mathcal{E}_K(\cdot, \cdot, \cdot)$. The experiment begins by uniformly selecting a key $K \leftarrow \mathcal{K}$. Then, when the adversary asks a query (N, A, M) , the oracle responds with $\mathcal{E}_K(N, A, M)$. The second term is the probability that $\mathcal{A}$ outputs 1 when given the oracle $\$(\cdot, \cdot, \cdot)$. This oracle, on input (N, A, M) , returns a uniformly random string of length $|M| + \tau$. We demand that $\mathcal{A}$ never asks two queries with the same first component (the N -value) and that it never asks a query outside of $\mathcal{N} \times \mathcal{A} \times \mathcal{M}$. Any such adversary query is invalid.

Next we define authenticity. With Π and $\mathcal{A}$ as before, let

$$\text{Adv}_{\Pi}^{\text{auth}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathcal{E}_K(\cdot, \cdot, \cdot)} \text{ forges}] .$$

Once again the experiment implicitly begins by selecting a key $K \leftarrow \mathcal{K}$. 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 $\mathcal{D}(K, N, A, C) \neq \perp$ yet there was no prior query (N, A, M) that returned C . We demand that $\mathcal{A}$ never asks two queries with the same N -value (the first component) and never asks a query outside of $\mathcal{N} \times \mathcal{A} \times \mathcal{M}$.

Informally, an AE scheme Π is secure if any adversary $\mathcal{A}$ with “reasonable” resources has “small” priv-advantage and small auth-advantage. Following the concrete-security tradition, and in view of the fact that we will only be defining OCB for use with an $n = 128$ bit blockcipher, we refrain from making any absolute or asymptotic definition of security.

An alternative definition of AE security merges privacy and authenticity to get a unified security measure

$$\text{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathcal{E}_K(\cdot, \cdot, \cdot), \mathcal{D}_K(\cdot, \cdot, \cdot)}] - \Pr[\mathcal{A}^{\$(\cdot, \cdot, \cdot), \perp(\cdot, \cdot, \cdot)}] .$$

The adversary is given either a pair of “real” encryption and decryption oracles or, alternatively, a pair of “fake” encryption and decryption oracles. The former begin by choosing a random $K \leftarrow \mathcal{K}$. After, the first oracle (the encryption oracle) responds to a query (N, A, M) with $\mathcal{E}(K, N, A, M)$, while the second oracle (the decryption oracle) responds to a query (N, A, C) with $\mathcal{D}(K, N, A, C)$. The $\$(\cdot, \cdot, \cdot)$ oracle, on query (N, A, M) , just returns $|M| + \tau$ uniformly random bits, where τ is the tag length of Π . The $\perp(\cdot, \cdot, \cdot)$ oracle, on query (N, A, C) , always returns $\perp$. The adversary may not repeat the first component, N , to its encryption oracle; it may not ask a decryption query of (N, A, C) following an encryption query of (N, A, M) that returned C ; and it may only ask encryption queries in $\mathcal{N} \times \mathcal{A} \times \mathcal{M}$ and decryption queries in $\mathcal{N} \times \mathcal{A} \times \mathcal{C}$.

The ae-security notion is essentially equivalent to the conjunction of priv-security and auth-security [41, Section 7]. In our proof of OCB’s security, we will use one direction of this claim.

We say that AE schemes $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and $\Pi' = (\mathcal{K}', \mathcal{E}', \mathcal{D}')$ *coincide* if their encryption and decryption algorithms compute identical functions: $\mathcal{E}(K, N, A, M) = \mathcal{E}'(K, N, A, M)$ and $\mathcal{D}(K, N, A, C) = \mathcal{D}'(K, N, A, C)$ for all K, N, A, M , and C . This implies $\mathcal{K} = \mathcal{K}'$, too.

We note that our priv and ae security notions demand indistinguishability from random bits (sometimes denoted IND\$) instead of indistinguishability from the encryption of random (or fixed) bits. This notion goes back to OCB1 [39]. There are several reasons for it. First, this notion of privacy is stronger, yet is achieved by OCB (and most other real-world AE schemes). At the same time, it is also what is most convenient to prove. Also, the encryption algorithm of an AE scheme meeting IND\$-security can be used as a pseudorandom generator (PRG), which might be convenient, in some cases, for the user. Perhaps most significantly, IND\$-security implies a form of anonymity [1] where one demands that ciphertexts be indistinguishable whether they are constructed from one hidden key K or are, instead, constructed using either of two hidden keys K_0, K_1 , which one being selected by the adversary.

3. Specification 1

We now define OCB (meaning OCB3). Given a blockcipher $E: \mathcal{K} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ and a tag length $\tau \in [0..128]$, Fig. 1 defines from E and τ the AE scheme $\Pi = \text{OCB}[E, \tau] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The nonce space $\mathcal{N} = \mathcal{N}_{\text{OCB}} = \{0, 1\}^{\leq 120}$ is the set of all binary strings with at most 120 bits. The message space $\mathcal{M} = \{0, 1\}^*$ and AD-space $\mathcal{A} = \{0, 1\}^*$ are all binary strings. The Setup procedure of Fig. 1 is implicitly run on or before the first call to $\mathcal{E}$ or $\mathcal{D}$. The variables it defines are understood to be global. While line 114 is written as an infinite loop, it is sufficient to compute $L[0], L[1], \dots, L[d]$ where 2^d upper bounds the number of 128-bit blocks in any plaintext M , ciphertext C , and associated data A . We write $\text{ntz}(i)$ for the number of trailing zeros in the binary representation of positive integer i (e.g., $\text{ntz}(1) = \text{ntz}(3) = 0$ and $\text{ntz}(4) = 2$). We write $\text{msb}(X)$ for the first (most significant) bit of the string X . We write $A \wedge B$ for the bitwise-*and* of the equal-length strings A and B . 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). We write either $A B$ or $A \parallel B$ for the concatenation of strings A and B . At line 121 we write $[a]_7$ for $a \in [0..127]$ encoded as a 7-bit string, while at line 126 we regard the variable Bottom as a number instead of a string.

For reasons of generality, OCB is defined to operate on bit strings. But for reasons of simplicity and efficiency, most implementations will assume that all strings operated on are strings of *bytes*, with a byte being eight bits.

Figure 2 illustrates encryption under OCB. Function Inc implicitly depends on the key K , with $\text{Inc}_i(\Delta) = \Delta \oplus L[\text{ntz}(i)]$, $\text{Inc}_\$ (\Delta) = \Delta \oplus L_\$$, $\text{Inc}_*(\Delta) = \Delta \oplus L_*$, and the (zero-argument) $\text{Init} = 0^{128}$. Here $L_* = E_K(0^{128})$, $L_\$ = 2 \cdot L_* = \text{double}(L_*)$, and $L[i] = 2^{2+i} \cdot L_*$ for all $i \geq 0$, the multiplication in $\text{GF}(2^{128})$. Here we write $2 = 0^{126}10 = \times$ to denote a particular nonzero point of the finite field. The diagrams should be read left to right and then top to bottom: first set Δ to be $\text{Init}_K(N)$; then modify Δ with Inc_1 ; then compute C_1 using the blockcipher E with pre- and post-whitening with Δ ; then modify Δ with Inc_2 ; and so on.

<pre> 100 algorithm $\mathcal{E}_K^{N,A}(M)$ 101 if $N \notin \mathcal{N}_{\text{OCB}}$ then return $\perp$ 102 $M_1 \cdots M_m M_* \leftarrow M$ where each 103 $M_i = 128$ and $M_* < 128$ 104 Checksum $\leftarrow 0^{128}$; $C \leftarrow \varepsilon$ 105 $\Delta \leftarrow \text{Init}_K(N)$ 106 for $i \leftarrow 1$ to m do 107 $\Delta \leftarrow \Delta \oplus L[\text{ntz}(i)]$ 108 $C \leftarrow E_K(M_i \oplus \Delta) \oplus \Delta$ 109 Checksum $\leftarrow \text{Checksum} \oplus M_i$ 10A if $M_* \neq \varepsilon$ then 10B $\Delta \leftarrow \Delta \oplus L_*$ 10C Pad $\leftarrow E_K(\Delta)$ 10D $C \leftarrow M_* \oplus \text{Pad}[1.. M_*]$ 10E Checksum $\leftarrow \text{Checksum} \oplus M_* 10^*$ 10F $\Delta \leftarrow \Delta \oplus L_\S$ 10G Final $\leftarrow E_K(\text{Checksum} \oplus \Delta)$ 10H Auth $\leftarrow \text{Hash}_K(A)$ 10I Tag $\leftarrow \text{Final} \oplus \text{Auth}$ 10J $T \leftarrow \text{Tag}[1..\tau]$ 10K return $C \parallel T$ 110 algorithm Setup(K) 111 $L_* \leftarrow E_K(0^{128})$ 112 $L_\S \leftarrow \text{double}(L_*)$ 113 $L[0] \leftarrow \text{double}(L_\S)$ 114 for $i \leftarrow 1, 2, \dots$ do $L[i] \leftarrow \text{double}(L[i-1])$ 115 return 120 algorithm Init$_K(N)$ 121 Nonce $\leftarrow [\tau \bmod 128]_7 \parallel 0^{120- N } \parallel 1 \parallel N$ 122 Top $\leftarrow \text{Nonce} \wedge 1^{122} 0^6$ 123 Bottom $\leftarrow \text{Nonce} \wedge 0^{122} 1^6$ 124 Ktop $\leftarrow E_K(\text{Top})$ 125 Stretch $\leftarrow \text{Ktop} \parallel (\text{Ktop} \oplus (\text{Ktop} \ll 8))$ 126 return (Stretch $\ll$ Bottom)[1..128] </pre>	<pre> 130 algorithm $\mathcal{D}_K^{N,A}(\mathcal{C})$ 131 if $N \notin \mathcal{N}_{\text{OCB}}$ or $\mathcal{C} < \tau$ then return $\perp$ 132 $C_1 \cdots C_m C_* T \leftarrow \mathcal{C}$ where each 133 $C_i = 128$ and $C_* < 128$ and $T = \tau$ 134 Checksum $\leftarrow 0^{128}$; $M \leftarrow \varepsilon$ 135 $\Delta \leftarrow \text{Init}_K(N)$ 136 for $i \leftarrow 1$ to m do 137 $\Delta \leftarrow \Delta \oplus L[\text{ntz}(i)]$ 138 $M \leftarrow D_K(C_i \oplus \Delta) \oplus \Delta$ 139 Checksum $\leftarrow \text{Checksum} \oplus M_i$ 13A if $C_* \neq \varepsilon$ then 13B $\Delta \leftarrow \Delta \oplus L_*$ 13C Pad $\leftarrow E_K(\Delta)$ 13D $M \leftarrow M_* \leftarrow C_* \oplus \text{Pad}[1.. C_*]$ 13E Checksum $\leftarrow \text{Checksum} \oplus M_* 10^*$ 13F $\Delta \leftarrow \Delta \oplus L_\S$ 13G Final $\leftarrow E_K(\text{Checksum} \oplus \Delta)$ 13H Auth $\leftarrow \text{Hash}_K(A)$ 13I Tag $\leftarrow \text{Final} \oplus \text{Auth}$ 13J $T' \leftarrow \text{Tag}[1..\tau]$ 13K if $T = T'$ then return M 13L else return $\perp$ 140 algorithm Hash$_K(A)$ 141 $A_1 \cdots A_a A_* \leftarrow A$ where each 142 $A_i = 128$ and $A_* < 128$ 143 Sum $\leftarrow 0^{128}$ 144 $\Delta \leftarrow 0^{128}$ 145 for $i \leftarrow 1$ to a do 146 $\Delta \leftarrow \Delta \oplus L[\text{ntz}(i)]$ 147 Sum $\leftarrow \text{Sum} \oplus E_K(A_i \oplus \Delta)$ 148 if $A_* \neq \varepsilon$ then 149 $\Delta \leftarrow \Delta \oplus L_*$ 140 Sum $\leftarrow \text{Sum} \oplus E_K(A_* 10^* \oplus \Delta)$ 14A return Sum 150 algorithm double(X) 151 return $(X \ll 1) \oplus (\text{msb}(X) \cdot 135)$ </pre>
---	--

Fig. 1. Definition of OCB $[E, \tau] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. Here, $E: \mathcal{K} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^n$ is a blockcipher and $\tau \in [0..128]$ is the tag length. Algorithms $\mathcal{E}$ and $\mathcal{D}$ are called with arguments $K \in \mathcal{K}$, $N \in \{0, 1\}^{\leq 120}$, and $M, C \in \{0, 1\}^*$. The nonce space for OCB is $\mathcal{N}_{\text{OCB}} = \{0, 1\}^{\leq 120}$.

4. Specification 2

In this section, we generalize OCB by describing the construction in terms of a tweakable blockcipher (TBC). We show that this alternative description really is a generalization of OCB by explaining how to recover the definition of OCB with a particular choice for its TBC.

Fig. 2. Illustration of OCB[E, τ]. The scheme depends on a blockcipher $E: \mathcal{K} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^n$ and the tag length $\tau \in [0..128]$. **Top:** Encrypting a plaintext M with a full final block. The checksum is $\text{Checksum} = M_1 \oplus M_2 \oplus M_3 \oplus M_4$. **Middle:** The plaintext M has a short final block, $1 \leq |M_*| < n$. The checksum is $\text{Checksum} = M_1 \oplus M_2 \oplus M_3 \oplus M_* \cdot 10^*$. **Bottom:** An AD of three full blocks (left) or two full blocks and one short block (right). **Throughout:** Offsets (the Δ -values) are updated and used top-to-bottom, then left-to-right. Offset initialization and increment functions return n -bit strings.

200 algorithm $\mathcal{E}_K^{N,A}(M)$	220 algorithm $\mathcal{D}_K^{N,A}(\mathcal{C})$
201 if $N \notin \mathcal{N}_{\text{OCB}}$ then return $\perp$	221 if $N \notin \mathcal{N}_{\text{OCB}}$ or $ \mathcal{C} < \tau$ then return $\perp$
202 $M_1 \cdots M_m M_* \leftarrow M$ where each	222 $C_1 \cdots C_m C_* T \leftarrow \mathcal{C}$ where each
203 $ M_i = n$ and $ M_* < n$	223 $ C_i = n$, $ C_* < n$, and $ T = \tau$
204 $\text{Checksum} \leftarrow 0^n$, $C_* \leftarrow \varepsilon$	224 $\text{Checksum} \leftarrow 0^n$, $M_* \leftarrow \varepsilon$
205 for $i \leftarrow 1$ to m do	225 for $i \leftarrow 1$ to m do
206 $C_i \leftarrow \tilde{E}_K^{N,i}(M_i)$	226 $M_i \leftarrow \tilde{D}_K^{N,i}(C_i)$
207 $\text{Checksum} \leftarrow \text{Checksum} \oplus M_i$	227 $\text{Checksum} \leftarrow \text{Checksum} \oplus M_i$
208 if $M_* = \varepsilon$ then $\text{Final} \leftarrow \tilde{E}_K^{N,m*}(\text{Checksum})$	228 if $C_* = \varepsilon$ then $\text{Final} \leftarrow \tilde{E}_K^{N,m*}(\text{Checksum})$
209 else $\text{Pad} \leftarrow \tilde{E}_K^{N,m*}(0^n)$	229 else $\text{Pad} \leftarrow \tilde{E}_K^{N,m*}(0^n)$
20A $C_* \leftarrow M_* \oplus \text{Pad}[1.. M_*]$	22A $M_* \leftarrow C_* \oplus \text{Pad}[1.. C_*]$
20B $\text{Checksum} \leftarrow \text{Checksum} \oplus M_* 10^*$	22B $\text{Checksum} \leftarrow \text{Checksum} \oplus M_* 10^*$
20C $\text{Final} \leftarrow \tilde{E}_K^{N,m*}(\text{Checksum})$	22C $\text{Final} \leftarrow \tilde{E}_K^{N,m*}(\text{Checksum})$
20D $\text{Auth} \leftarrow \text{Hash}_K(A)$	22D $\text{Auth} \leftarrow \text{Hash}_K(A)$
20E $\text{Tag} \leftarrow \text{Final} \oplus \text{Auth}$	22E $\text{Tag} \leftarrow \text{Final} \oplus \text{Auth}$
20F $T \leftarrow \text{Tag}[1..\tau]$	22F $T' \leftarrow \text{Tag}[1..\tau]$
20G return $C_1 \cdots C_m C_* \parallel T$	22G if $T = T'$ then return $M_1 \cdots M_m M_*$
	22H else return $\perp$
210 algorithm $\text{Hash}_K(A)$	
211 $\text{Sum} \leftarrow 0^n$	
212 $A_1 \cdots A_a A_* \leftarrow A$ for $ A_i = n$, $ A_* < n$	
213 for $i \leftarrow 1$ to a do	
214 $\text{Sum} \leftarrow \text{Sum} \oplus \tilde{E}_K^{i*}(A_i)$	
215 if $A_* \neq \varepsilon$ then	
216 $\text{Sum} \leftarrow \text{Sum} \oplus \tilde{E}_K^{a*}(A_* 10^*)$	
217 return Sum	

Fig. 3. Definition of $\Theta\text{CB}[\tilde{E}, \tau]$. Here $\tilde{E}: \mathcal{N} \times \mathcal{T}_{\Theta\text{CB}} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ is a tweakable blockcipher and $\tau \in [0..n]$ is the tag length. We have that $\text{OCB}[E, \tau] = \Theta\text{CB}[\tilde{E}, \tau]$ when $\tilde{E} = \text{Tw}[E, \tau]$.

4.1. The TBC-Based Scheme

We can view OCB as an AE scheme built not from a blockcipher but from a tweakable blockcipher (TBC) [30], as defined and discussed in Section 2. Refer to Fig. 3, which defines the TBC-based version of OCB, which we name ΘCB . It depends on a tweakable blockcipher $\tilde{E}: \mathcal{K} \times \mathcal{T}_{\Theta\text{CB}} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ with $n = 128$ and a tag length $\tau \in [0..128]$. The tweak space for $\tilde{E}$ is the rather complicated set

$$\mathcal{T}_{\Theta\text{CB}} = \mathcal{N} \times \mathbb{N}_1 \cup \mathcal{N} \times \mathbb{N}_0 \times \{*\} \cup \mathcal{N} \times \mathbb{N}_0 \times \{\$\} \cup \mathcal{N} \times \mathbb{N}_0 \times \{*\$\} \cup \mathbb{N}_1 \cup \mathbb{N}_0 \times \{*\}$$

where $\mathcal{N} = \mathcal{N}_{\text{OCB}} = \{0, 1\}^{\leq 120}$ is the nonce space for OCB and $\mathbb{N}_1$ and $\mathbb{N}_0$ are the positive and nonnegative integers, respectively. For the TBC-based version of OCB, it is not actually necessary that $n = 128$, and the set $\mathcal{T}_{\Theta\text{CB}}$ could just as well be defined from any nonce space $\mathcal{N}$.

Tweaks, it can be seen, are of six mutually exclusive types. Those of the first type are in the set $\mathcal{B} = \mathcal{N} \times \mathbb{N}_1$ of *bidirectional* tweaks. Omitting parenthesis and commas when writing tweaks, calls to the TBC $\tilde{E}$ will look like $\tilde{E}_K^{N,i}(X)$, $\tilde{E}_K^{N,i*}(X)$, $\tilde{E}_K^{N,i\$}(X)$, $\tilde{E}_K^{N,i*\$}(X)$, $\tilde{E}_K^{i*}(X)$, or $\tilde{E}_K^{i*\$}(X)$.

With $\tilde{E}: \mathcal{K} \times \mathcal{T}_{\Theta\text{CB}} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ and $\tau \in [0..n]$ the AE scheme $\Pi = \Theta\text{CB}[\tilde{E}, \tau] = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ continues to use the nonce space $\mathcal{N}_{\text{OCB}}$ of OCB, its message space is $\mathcal{M} = \{0, 1\}^*$, its AD space is $\mathcal{A} = \{0, 1\}^*$, and its ciphertext space is $\mathcal{C} = \{0, 1\}^*$ (actual ciphertexts are, as before, in $\{0, 1\}^{\geq \tau}$).

Fig. 4. Illustration of ΘCB . Scheme $\Theta\text{CB}[\tilde{E}, \tau]$ depends on a TBC $\tilde{E}: \mathcal{K} \times \mathcal{T}_{\Theta\text{CB}} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a tag length $\tau \in [0..n]$. The top figure shows the treatment of a plaintext M having a full final block ($|M_4| = n$, $\text{Checksum} = M_1 \oplus M_2 \oplus M_3 \oplus M_4$); the middle picture shows the treatment of a plaintext M having a short final block ($1 \leq |M_*| < n$, $\text{Checksum} = M_1 \oplus M_2 \oplus M_3 \oplus M_*10^*$). The bottom-left picture shows the processing of a three-block AD; on bottom-right, an AD with a short final block. Algorithm $\Theta\text{CB}[E, \tau]$ coincides with $\Theta\text{CB}[\tilde{E}, \tau]$ for a particular TBC $\tilde{E} = \text{Tw}[E, \tau]$ built from E .

Encryption under ΘCB is illustrated in Fig. 4. The encryption process is conceptually simpler than it is under OCB because the offsets (the Δ -values) of Fig. 2 are now gone, replaced by the tweaks that, conceptually, identify unrelated permutations E_K^T and $E_K^{T'}$ for distinct tweaks T, T' .

$\tilde{E}_K^{N^i}(X) = E_K(X \oplus \Delta) \oplus \Delta$ with $\Delta = \text{Initial} \oplus \lambda_i L$	for $i \geq 1$
$\tilde{E}_K^{N^{i*}}(X) = E_K(X \oplus \Delta)$	with $\Delta = \text{Initial} \oplus \lambda_i^* L$ for $i \geq 0$
$\tilde{E}_K^{N^{i\$}}(X) = E_K(X \oplus \Delta)$	with $\Delta = \text{Initial} \oplus \lambda_i^{\$} L$ for $i \geq 0$
$\tilde{E}_K^{N^{i*\$}}(X) = E_K(X \oplus \Delta)$	with $\Delta = \text{Initial} \oplus \lambda_i^{*\$} L$ for $i \geq 0$
$\tilde{E}_K^i(X) = E_K(X \oplus \Delta)$	with $\Delta = \lambda_i L$ for $i \geq 1$
$\tilde{E}_K^{i*}(X) = E_K(X \oplus \Delta)$	with $\Delta = \lambda_i^* L$ for $i \geq 0$

where

Nonce = $[\tau \bmod 128]_7 \parallel 0^{120- N } \parallel 1 \parallel N$	$L = E_K(0^{128})$
Top = $\text{Nonce} \wedge 1^{122} 0^6$	$\gamma_0 = 0$ //Gray code seq: 0, 1, 3, 2, 6, 7, 5, 4, 12, ...
Bottom = $\text{Nonce} \wedge 0^{122} 1^6$	$\gamma_i = \gamma_{i-1} \oplus 2^{\text{ntz}(i)}$ if $i \geq 1$
Ktop = $E_K(\text{Top})$	$\lambda_i = 4 \gamma_i$
Stretch = $\text{Ktop} \parallel (\text{Ktop} \oplus (\text{Ktop} \ll 8))$	$\lambda_i^* = 4 \gamma_i + 1$
Initial = $(\text{Stretch} \ll \text{Bottom})[1..128]$	$\lambda_i^{\$} = 4 \gamma_i + 2$
	$\lambda_i^{*\$} = 4 \gamma_i + 3$

Fig. 5. Definition of $\tilde{E} = \text{Tw}[E, \tau]$, the tweakable blockcipher built from E . Multiplication is in the finite field $\text{GF}(2^{128})$. We identify field points and 128-bit strings, field-point addition and xor.

4.2. Instantiating the TBC

We can instantiate the TBC of $\Theta\text{CB}[\tilde{E}, \tau]$ in a way that makes the scheme coincide with $\text{OCB}[E, \tau]$. With $n = 128$ and $\mathcal{N} = \mathcal{N}_{\text{OCB}} = \{0, 1\}^{\leq 120}$, map the blockcipher $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ to a TBC $\tilde{E} = \text{Tw}[E, \tau]$ with signature $\tilde{E}: \mathcal{K} \times \mathcal{T}_{\Theta\text{CB}} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$. The construction is specified in Fig. 5. There, multiplication is in $\text{GF}(2^{128})$ using the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$. We do not distinguish between strings and what they represent in this finite field.

The code of Fig. 5 is more understandable with the following background. The sequence of values $\gamma_0, \gamma_1, \gamma_2, \dots$ employed is called a *Gray code*. We give a couple of well-known facts about Gray codes. First, that $\gamma: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is a permutation. In fact, it is a permutation on $\mathbb{Z}_{2^i}$ for each i . Thus, we also have that $0 \leq \gamma_i \leq 2i$ for all i . It follows from these facts and the definition of our λ -values that all of the values $\Lambda = \{\lambda_i, \lambda_i^*, \lambda_j^{\$}, \lambda_j^{*\$}: 1 \leq i \leq 2^{120}, 0 \leq j \leq 2^{120}\}$ are distinct and nonzero. This is the only thing we will actually need of these values.

The reader can now check that $\text{OCB}[E, \tau] = \Theta\text{CB}[\text{Tw}[E, \tau], \tau]$ —we have defined Tw to make that so—a fact important enough to record as a lemma:

Lemma 1. *For any blockcipher $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ and number $\tau \in [0..n]$, the construction $\text{OCB}[E, \tau]$ coincides with $\Theta\text{CB}[\text{Tw}[E, \tau], \tau]$.*

5. Evolution

We now explain some of the design choices made for OCB, simultaneously sketching the mode's history.

DEVELOPMENT OF OCB1. The initial reason for developing OCB1 [39] was to improve the efficiency of authenticated encryption. Prior work by Jutla [24], and also by Katz

and Yung [28] and Gligor and Donescu [11], had evidenced that tightly integrating privacy and authenticity let one get by with less work than what one would see from generically composing separate privacy and authenticity mechanisms. Attending to “low-level” concerns, we aimed to create a blockcipher-based AE scheme that would push the efficiency bar. OCB1 handles plaintexts of any length and produces ciphertexts τ -bits longer. It uses a near-minimal number of blockcipher calls and minimizes work beyond those calls. It allows nearly all blockcipher calls to be performed in parallel. It assumes a nonce, not a random IV. It is deterministic. A single key underlies all blockcipher calls. It is online, making a single pass over the plaintext or ciphertext, whose length may not be known in advance.

DEVELOPMENT OF OCB2. A series of devastating attacks on 802.11 (WiFi) security created an urgent need to replace its confidentiality mechanism. Jesse Walker, Nancy Cam-Winget, and others advocated replacing it with OCB1. But they explained to Rogaway that there would need to be some way to authenticate but not encrypt header information. Formalizing and achieving this end led to the idea of *associated data* and to the paper by Rogaway that introduced it [37].

A couple years later, a different paper by Rogaway [38] proposed adjusting the offsets used in OCB1, going from the Gray-code ordered $L, 3L, 2L, 6L, 7L, 5L, \dots$ to the powering-up ordered $L, 2L, 2^2L, 2^3L, 2^4L, \dots$. OCB2 does this, and also accommodates the AD. It was adopted as an ISO standard in 2009 [18]. As mentioned already, the scheme has a major error [17].

OCB1’S SPEED QUESTIONED. In 2004 McGrew and Viega wrote a paper claiming that GCM [10] was about as fast, if not faster, than OCB1 [31]. They suggested that two design issues underlie this. First, OCB1 uses $m + 2$ blockcipher calls to encrypt a message of $m = \lceil |M|/128 \rceil$ blocks. In contrast, GCM makes do with $m + 1$. Second, OCB1 twice needs the result of one AES computation before another AES computation can proceed. Both in hardware and in software, this can degrade performance. Still, we suspected that these two matters would have only a minor impact on performance. The timing studies were misleading, we believed, because McGrew and Viega had compared a reference implementation of OCB with an optimized implementation of GCM. Even that was irreproducible due to McGrew and Viega’s use of proprietary GCM code.

We decided to revise OCB1/OCB2 to address and assess the two efficiency concerns above. We integrated scheme development and performance measurements. The result of our work was the definition of OCB3 and a comparative study of the performance of CCM [9], GCM, and the OCB triplet [26]. We found that, across a variety of platforms, all versions of OCB were considerably faster than both CCM and GCM. The performance differences among OCB1, OCB2, and OCB3 were small. OCB3 had the best performance, but OCB2’s performance was actually worse than OCB1’s. We now look at some of the OCB3 changes over OCB1/OCB2.

REDUCED LATENCY. Suppose that the message $M = M_1 \cdots M_m M_*$ being encrypted is not a multiple of 128 bits: it has a final block M_* of 1–127 bits. With OCB1 and OCB2 one makes an M_m -dependent blockcipher call to compute a ciphertext block C_m that is in turn used to create a Checksum that is enciphered with another blockcipher call. This can result in a pipeline stall [31]: one blockcipher call must finish before the next one can begin. For OCB3 we restructure the algorithm so that the Checksum does not

depend on any ciphertext block: we simplify the Checksum to $M_1 \oplus M_2 \oplus \dots \oplus M_m$ when there is a full final block M_m , and $M_1 \oplus M_2 \oplus \dots \oplus M_m \oplus M_*10^*$ otherwise.

INCREMENTING OFFSETS. In OCB1, each noninitial offset is computed from the prior one by xoring in some key-derived value: the i th offset is $\Delta \leftarrow \Delta \oplus L[\text{ntz}(i)]$. In OCB2, each noninitial offset is computed from the prior one by multiplying it, in $\text{GF}(2^{128})$, by a constant: $\Delta \leftarrow (\Delta \ll 1) \oplus (\text{msb}(\Delta) \cdot 135)$, an operation we think of as *doubling*. The first approach turns out to be faster [39]. 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 cannot 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, 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. For all of these reasons, OCB3 reverts to the OCB1 method incrementing offsets.

In the development of OCB3, we tried a variety of other methods to generate the needed offsets, exploring word-oriented LFSRs (linear feedback shift registers) of various designs [39, Appendix B]. No alternative performed significantly better, even in isolation, than the method used in OCB1.

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 ; and one to encipher the final Checksum. The first of these is easy to eliminate if one is willing to replace a blockcipher call by, say, $K_1 \cdot N$, the product in $\text{GF}(2^{128})$ of nonce N and a variant K_1 of K . But such a change would necessitate implementing a $\text{GF}(2^{128})$ multiply. We use a different method to trim the extra blockcipher call, most of the time, for a counter-based nonce: we employ a new xor-universal hash function, which we call the *stretch-then-shift* hash. The initial offset Δ is $\Delta = H_K(N) = (\text{Stretch} \ll \text{Bottom})[1..128]$ 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 description ignores the inclusion of the tag length τ along with N .) The technique ensures that when the nonce N is a counter, the initial offset Δ can be computed without a new blockcipher call about $63/64 \approx 98\%$ of the time. In this way we reduce cost from $m + 2$ blockcipher calls to an amortized $m + 1.02$ blockcipher calls. Note, however, that achieving this reduction requires an implementation to cache the value of K_{top} (line 124) and to notice, when true, that it need not be recomputed. And there are security considerations in doing this, for it is essential that the adversary not have access to the cached value.

INCORPORATING TAG LENGTH. The algorithm of this paper coincides with that of RFC 7253 [27] but differs from the algorithm in the proceedings paper [26] in one minor way: the tag length is used in computing the initial offset Δ (line 121). A change in this direction was requested during the RFC review process in order that a ciphertext created for one value of the tag length τ should not be of value in forging a ciphertext for a different value of τ . The change is heuristic, as security claims associated with OCB assume τ to be fixed. Subsequent work by Reyhanitabar, Vaudenay, and Vizár

demonstrated that the manner in which we incorporate τ does not work to improve the security notion to one where the same key is used for multiple tag lengths [36].

REPRESENTING FIELD POINTS. A low-level choice where OCB and GCM [10] part ways is in the representation of field elements. In GCM the polynomial $a_{127}x^{127} + \dots + a_1x + a_0$ corresponds to the string $a_0 \dots a_{127}$ rather than the string $a_{127} \dots a_0$. The usual convention on machines, whether big-endian and little-endian, is that the msb is the leftmost bit of any register. One advantage of following that convention is that a left shift by one can be implemented by adding a register to itself, an operation that is sometimes faster than a 1-bit logical shift. For example, on an ARM Cortex-A57 addition has one cycle latency but logical shift has two.

INCREMENTAL APIS. Unlike OCB1 and OCB2, under OCB3, each 128-bit block of plaintext is processed in the same way whether or not it is the final 128 bits. This change facilitates implementing an incremental API, as 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 over. Similarly, each 128-bit block of AD is now treated the same way whether it is or is not the message's end, simplifying the incremental provisioning of it.

ENDIAN ISSUES. While we expect the majority of processors running OCB will be little-endian, the mode's definition does nothing to favor this convention. Endian issues arise when "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. One reason for this is the efficient byte-reversal capability on most modern processors (e.g., the `pshufb` instruction can reverse 16 bytes on x86 machines in a single cycle). Also, the OCB1-based approach for incrementing offsets allows precomputed values to be endian-adjusted at key-setup time, removing most endian-dependency for subsequent encryption or decryption calls. Since it makes little difference to performance, and big-endian specifications are conceptually easier, OCB does not make any gestures toward little-endian orientation.

6. Performance

Figures 6, 7, and 8 compare the performance of OCB and several other AE schemes. The OCB alternatives were chosen based on their popularity and salience. The TLS 1.3 specification [35] mentions three AE algorithms—GCM [10,31], CCM [9,43] and Chacha20-Poly1305 [5,32]. These schemes are well established and probably represent the bulk of AE in use today. So we begin with these three schemes. Then, in the recently concluded CAESAR competition, AEGIS-128 [44] and OCB were winners of the "high-performance" category. So we include AEGIS-128 as a fourth point of comparison. Finally, on the 64-bit Intel architecture we include AES-GCM-SIV, as specified in RFC 8452 [13] (2018). The AES-GCM-SIV mechanism builds on Rogaway and Shrimpton's SIV [41] to add nonce-reuse misuse-resistance, the most important security

Fig. 6. Performance of AE schemes on x86-64. The graphs show CPU cycles-per-byte (vertical axis, lower is better) for encrypting messages of various lengths (horizontal axis) on an Intel Core i3-8121U “Cannon Lake” processor .

property, in our view, that is *not* delivered by OCB or the other alternatives we have named. AES-GCM-SIV is benchmarked only on the 64-bit Intel architecture because, at the time of writing, that is the only high-quality implementation available.

We ran experiments on three different architectures. The most important architectures today are 64-bit Intel x86 for servers, desktops and laptops, and 64-bit ARM for high-performance embedded applications such as smartphones and tablets. We therefore benchmark on CPUs fitting these descriptions. For 64-bit Intel we use a recent Intel Core i3-8121U (“Cannon Lake”). For 64-bit ARM we test on a high-performance Cortex-A73. As a catch-all for architectures not falling into either of these categories, we also benchmark on an older 32-bit ARMv5 processor (Marvell 88FR131 “Feroceon”). The Feroceon results capture how much work each algorithm needs when the unit of work is a 32-bit RISC assembly instruction and the processor lacks hardware acceleration such as an AES unit or vector registers. The Feroceon is also a reasonable stand-in for what can be expected from ARM’s current M-series embedded processors.

SOURCE CODE. A problem sometimes seen in performance comparisons is to include implementations of uneven quality. To avoid this, we considered for benchmarking the best-performing version of each scheme among several sources. OpenSSL, Libgcrypt, and WolfSSL all provide open-source cryptographic libraries with significant assembly-language acceleration targeting multiple architectures. For each algorithm implemented by each library, we identified its peak speed on multiple architectures. In every case OpenSSL’s library was the fastest. This is not surprising: OpenSSL has a longer record of producing good-quality assembly than the others. As a sanity check, whenever possible

Fig. 7. Performance of AE schemes on 64-bit ARM. The graphs show CPU cycles-per-byte (vertical axis, lower is better) for encrypting messages of various lengths (horizontal axis) on an ARM Cortex-A73 processor.

Fig. 8. Performance of AE schemes on a low-end 32-bit embedded processor. The graphs show CPU cycles-per-byte (vertical axis, lower is better) for encrypting messages of various lengths (horizontal axis) on a Marvell "Feroceon" ARMv5 processor.

we also verified that speeds reported on the SUPERCOP benchmarking website [42] were not faster than the speeds we found.

The AEGIS-128 implementation we used for benchmarking is a modified version of the optimized one placed on the SUPERCOP benchmarking website by the AEGIS designers. The AES-GCM-SIV implementation we used is from Google's BoringSSL library. We ourselves have supplied the OCB code. The AEGIS-128 and OCB codes were written in C (with AES intrinsics when available) and all other AE algorithms have their kernels written in assembly. All non-library code was compiled with both Clang 8.0 and GCC 8.3 or 9.1, using both optimization levels 2 and 3, and whichever combination performed best is reported. The OpenSSL and BoringSSL libraries were built using the library's build script with the addition of architecture-specific flags. We found that changing which compiler or optimization level was used to compile the libraries made no measurable difference in our tests, no doubt because the time-critical code was already in assembly, thus unaffected by compiler options.

DISCUSSION. Having AES in hardware is the most significant determinant of performance. On our 64-bit ARM and 64-bit Intel processors—both of which have hardware AES support as well as support for GCM's "carryless" multiplication—OCB, AEGIS, and GCM are fastest. On these architectures, OCB and AEGIS have similar performance profiles, while GCM encrypts 2KB messages 40–60% slower. Chacha20-Poly1305 and CCM have peak speeds 3–7 times slower than OCB and AEGIS-128. On systems without AES in hardware, Chacha20-Poly1305 is the clear winner. On our Feroceon system Chacha20-Poly1305 encrypts 2KB messages 18% faster than AEGIS-128, twice as fast as OCB, and over three times faster than CCM and GCM.

The relative speeds of these algorithms are easy to explain. OCB and AEGIS-128 were designed to have little marginal overhead beyond calls to the AES round function. On systems with a hardware AES unit, this results in outstanding peak performance. Although CCM and GCM can also take advantage of a hardware AES unit, the computation of GHASH for GCM and the CBC-MAC for CCM, in addition to the counter-mode encryption each performs, is enough to separate them from the speed leaders. Chacha20-Poly1305 does not use AES or carryless multiplication, instead depending on integer addition, rotation, and xor; it gets no benefit from AES hardware support. Chacha20-Poly1305 is designed to benefit from vectorized instructions, like AVX on Intel and NEON on ARM, but these do not have enough accelerative power to keep up with the AES unit's capabilities.

On a system without an AES unit, we see that AEGIS-128 is about twice as fast as OCB (because it computes about half as many AES rounds, per byte of input, as OCB), and OCB is about twice as fast as CCM (because it computes about half as many AES rounds as CCM). GCM is somewhere between OCB and CCM because GCM substitutes a less expensive GHASH computation for CCM's CBC-MAC. The simplicity of the Chacha20-Poly1305 design leads to its superior performance on simpler systems.

EFFECT OF PARALLELISM. On systems with an AES unit in hardware, parallelism has a profound effect. OCB can take advantage of any amount of AES parallelism. If a system is able to compute, say, 32 AES calls in parallel, then OCB computation can be arranged to loop and process 32 blocks per iteration. This means that OCB's performance depends primarily on how many AES computations can be done in parallel and how long each

takes to complete. On Intel's Cannon Lake architecture, its AES unit can issue two AES-rounds per cycle and they take five cycles to complete. This means that if the pipeline can be kept full, it can complete ten rounds of processing every five cycles. Since each 16 bytes of plaintext requires at least ten rounds of AES processing, this implies a maximum throughput of five cycles per 16 bytes, or 0.31 cpb. On Intel's Skylake architecture, where only one AES round can be issued per cycle, it will take at least ten cycles per 16 bytes, so 0.62 cpb. OCB achieves this lower-bound speed on Skylake but not on Cannon Lake. There is enough slack in the functional units during Skylake's computation to absorb the computational overhead beyond AES computations, whereas on Cannon Lake there is not.

Looking toward newer architectures, Intel is currently rolling out its AVX-512 architecture. Our Cannon Lake testbed has some previously developed 512-bit vector instructions, which are used by Chacha20-Poly1305, but Intel's Ice Lake architecture brings more significant improvements. Its AES unit doubles throughput and shaves 25% off latency which will benefit OCB much more than the other AE algorithms. AEGIS-128 has a serial component—it cannot start the first AES round of an input block until the first AES round of the prior block has completed—implying that its performance is limited by AES round latency. This should result in a 25% speed increase between Cannon Lake and Ice Lake architectures, but that is less than the potential doubling of OCB speed. GCM will see a doubling in AES throughput, but carryless multiplication will maintain its throughput, limiting any performance increase. CCM's serial CBC-MAC computation will be 25% faster on Ice Lake than on Cannon Lake, but will continue to keep CCM's performance slowest of the group. Chacha20-Poly1305's AVX-512 speed improvements are already reflected in this study and should see no significant improvement on Ice Lake. Preliminary results on an Ice Lake system, completed as we go to press, show an OCB peak throughput of 0.19 cpb, which is very close to the predicted doubling.

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 2048 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: the `clock` function of C and the 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 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, 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 it is fast. 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. To reduce the impact of these problems, we read the TSC once before and after encrypting the same message 65 times and 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.

7. Security

The key steps for proving OCB secure are establishing the security of the Θ CB construction when based on an idealized TBC, and then showing that our way of creating a tweakable blockcipher, the Tw-construction, works. Both steps are in the information-theoretic setting. Passing to the complexity-theoretic setting for the final result is then standard.

7.1. Security of the TBC-Based Scheme

The following lemma establishes the priv-security and auth-security of Θ CB when based on an ideal TBC.

Lemma 2. Fix $n = 128$ and $\tau \in [0..n]$ and $\tilde{\mathbb{E}} = \tilde{\mathbb{E}}^{\mathcal{T}_{\Theta\text{CB}}, n}$. Let $\Pi = \Theta\text{CB}[\tilde{\mathbb{E}}, \tau]$. Then,

- $\text{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}_{\text{priv}}) = 0$ for any adversary $\mathcal{A}_{\text{priv}}$, while
- $\text{Adv}_{\Pi}^{\text{auth}}(\mathcal{A}_{\text{auth}}) \leq 2^{n-\tau} / (2^n - 1) \leq 1.1/2^{\tau}$ for any adversary $\mathcal{A}_{\text{auth}}$.

Note that in this idealized setting, privacy and authenticity do not degrade with message lengths, AD lengths, or the number of queries: privacy is perfect, while the chance of forging is about $1/2^{\tau}$.

Proof. Consulting Fig. 4 will make the proof more understandable. Keep in mind that, in the idealized setting we consider, $\tilde{E}_K^{Ni}$, $\tilde{E}_K^{Ni\$}$, $\tilde{E}_K^{Ni*}$, $\tilde{E}_K^{Ni*\$}$, $\tilde{E}_K^i$ and $\tilde{E}_K^{i*}$ are random permutations on n bits (for all permissible N and i). To emphasize this, we write π in place of $\tilde{E}_K$, so π^{Ni} , π^{Ni*} , $\pi^{Ni\$}$, $\pi^{Ni*\$}$, π^i and π^{i*} . Any two such permutations are independent of one another as long as their superscripts are distinct. We must argue privacy and authenticity.

PRIVACY. During the adversary's attack it asks queries $(N^1, A^1, M^1), \dots, (N^q, A^q, M^q)$ and gets responses $\mathcal{C}^1, \dots, \mathcal{C}^q$ where $\mathcal{C}^j = C^j \parallel T^j$ and $C^j = C_1^j \dots C_{m_j}^j$ or $C^j = C_1^j \dots C_{m_j}^j C_*^j$. In general, in this proof we will use a superscript of j on a variables to indicate the value of that variable following the adversary's j th query. We may do this with any variable appearing in Fig. 4.

Since the N^j -values must be distinct, each permutation of the form $\pi^{N^j \dots}$ is used at most once. We are thus applying independent random permutations to a single point each, so all of these outputs are uniformly random and independent. The bulk of what is returned to the adversary, all the C_i^j values, are such $\pi^{N^j i}$ outputs. But the C_*^j and T^j values are not just permutation outputs but are, instead, $\pi^{N^j m^j *}$ or $\pi^{N^j m^j \$}$ or $\pi^{N^j m^j *\$}$ outputs that are xored with $M_*^j 0^*$ and truncated to $|M_*^j|$ bits, or, alternatively, are xored

with Auth^j and then truncated to τ bits. Either way, the result remains uniform and independent of all other outputs, as Auth^j , M_*^j , and τ are independent of $\pi^{N^j \dots}$ outputs. We conclude that each result from the adversary's j^{th} query is a uniformly random string $\mathcal{C}^j$ of length $|M^j| + \tau$ that is independent of all other query responses. It follows 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 Tag is *not* some particular value Tag_0 . All of the $2^n - 1$ other possibilities are equally likely. Then, your chance of correctly predicting the τ -bit prefix of Tag is at most $2^{n-\tau}/(2^n - 1)$. That is because the best strategy is to guess any τ -bit string other than the τ -bit prefix of Tag_0 . The probability of being right under this strategy is $2^{n-\tau}/(2^n - 1)$. We will repeatedly use this fact in the sequel.

Now suppose that the adversary asks a sequence of queries $(N^1, A^1, M^1), \dots, (N^q, A^q, M^q)$ and then makes its forgery attempt $(N, A, \mathcal{C})$, where $\mathcal{C} = C \parallel T$, $C = C_1 \dots C_c$ or $C = C_1 \dots C_c C_*$. We need to bound the probability that $(N, A, \mathcal{C})$ is a successful forgery.

We begin by considering the case where the forgery attempt $(N, A, \mathcal{C})$ employs a nonce N that is not among $\{N^1, \dots, N^q\}$. Then to forge the adversary needs to find the correct value of $T = (\pi^{N^{\dots}}(\text{Checksum}) \oplus \text{Auth})[1..\tau]$ but has seen nothing that depends on $\pi^{N^{\dots}}$. Having no relevant information to use, the adversary's chance of success is clearly $2^{-\tau}$, which is less than $2^{n-\tau}/(2^n - 1)$.

Given the above, we can assume that the forgery attempt $(N, A, \mathcal{C})$ uses a nonce $N = N^j$ that coincides with a nonce from some prior query. All the other queries had to employ different nonces and thereby generated information theoretically unrelated to the adversary's task at hand. We can therefore disregard these other encryption queries and assess the maximal forging probability for a new and simpler game: the adversary asks a single encryption query (N, A', M') , getting a response $\mathcal{C}' = C' \parallel T'$, and then it tries to forge, with the same nonce, outputting a triple $(N, A, \mathcal{C})$. We must bound the probability that such a forgery is valid. We write $A' = A'_1 \dots A'_{a'}$, or $A' = A'_1 \dots A'_{a'} A'_*$, $M' = M'_1 \dots M'_{m'}$, or $M' = M'_1 \dots M'_{m'} M'_*$, $A = A_1 \dots A_a$ or $A = A_1 \dots A_a A_*$, $\mathcal{C} = C \parallel T$, $C = C_1 \dots C_c$ or $C = C_1 \dots C_c C_*$. We proceed by case analysis, relating the form of the forgery attempt $(N, A, \mathcal{C})$ to the encryption query (N, A', M') . We say that a string is *full* if it has a multiple of n bits, and *partial* otherwise.

Case 1: $A \neq A'$, $A \neq \varepsilon \neq A'$. We have several subcases to consider. *Case 1a.* If A' is full and A is partial, then $\text{Auth}' \parallel \text{Auth}$ will be U_{2n} distributed, the uniform distribution on $2n$ bits, and the adversary will only be able to forge with probability $2^{-\tau}$. The uniformity of $\text{Auth}' \parallel \text{Auth}$ is because Auth' will depend on a π^a output but no π^{a*} output, while Auth will depend on a π^{a*} output. The case when A' is partial and A is full is similar. *Case 1b.* If A' and A are full and $a \neq a'$, then $\text{Auth}' \parallel \text{Auth}$ will again be U_{2n} distributed, whence the forging probability will be $2^{-\tau}$. The uniformity of $\text{Auth}' \parallel \text{Auth}$ is because each includes a π^1 output in the xor but only one xors in a $\pi^{\max(a, a')}$ output. The case of A' and A being partial but $a \neq a'$ is analogous. *Case 1c.* Suppose $A' = A'_1 \dots A'_a$ and $A = A_1 \dots A_a$ are full, equal in length, but distinct; say $A'_j \neq A_j$ for some j . Then, even if the adversary were given not only $\mathcal{C}'$ but also Final, all $\pi^i(A'_i)$ values, and all

$\pi^i(A_i)$ values except for $i = j$, still the value of $\text{Auth} = \oplus_i \pi^i(A_i) = Z \oplus \pi^j(A_j)$, and therefore Tag , would be uniform in a space of $2^n - 1$ possible values. We are now in the setting of game G , and the adversary's chance to predict T , the first τ bits of Tag , is at most $2^{n-\tau}/(2^n - 1)$. *Case 1d.* The last case is when $A' = A'_1 \dots A'_a A'_*$ and $A = A_1 \dots A_a A_*$ are partial, equal in length, but distinct. If $A'_j \neq A_j$ for some j , then proceed as with Case 1c. If $A'_* \neq A_*$, then $A'_* 10^* \neq A_* 10^*$ and proceed as with Case 1c, but with $\pi^{a*}(A_* 10^*)$ the unknown value.

Case 2: $A \neq A'$, $A = \varepsilon$ or $A' = \varepsilon$. If $A' = \varepsilon \neq A$, then $\text{Auth}' = 0^n$, while Auth is U_n distributed and independent of Final , so Tag is uniform and independent of all values the adversary has observed, whence its ability to predict its τ -bit prefix is at most $2^{-\tau}$. If $A' \neq \varepsilon = A$ then $\text{Auth} = 0^n$ and the adversary must output the τ -bit prefix of Final . It has received no information on any $\pi^{Ni}\$$ or $\pi^{Ni*}\$$ from its encryption query, as the only such output was masked with Auth' , formed using a $\pi^{a'}$ or $\pi^{a'*}$ output. So the forging probability is again $2^{-\tau}$.

Case 3: $A = A'$; M' partial and C full, or the other way around. For this and all remaining cases, we imagine providing the adversary with $\text{Auth}' = \text{Auth}$. So the adversary must predict the first τ bits of Final . But Final is the output of a random permutation $\pi^{Nc}\$$ that has not yet been applied to any point. With no relevant information for carrying out this task, its chance to forge is at most $2^{-\tau}$. For the case with M' full and C partial, Final is, similarly, an output $\pi^{Nc*}\$$ where this permutation has not yet been applied to any point.

Case 4: $A = A'$; $c \neq m'$; M' and C both full or both partial. the adversary must predict the first τ bits of Final . But Final is the output of a random permutation $\pi^{Nc}\$$ (if M' and C both full) that has not yet been used at even one point. With no relevant information, the adversary forges with probability at most $2^{-\tau}$. Similarly when M' and C are both partial.

Case 5: $A = A'$, $c = m'$, M' and C full. For the forgery to be valid we must have $C' \neq C$ so let j to be the smallest value where $C'_j \neq C_j$. Assume the adversary is given the value of $\text{Auth}' = \text{Auth}$, all π^{Ni} permutations where $i \neq j$, and all $\pi^{\dots}\$$ permutations. Then the adversary will know $\text{Checksum}'$ and all the values but one that xored together make Checksum . But it will be missing one addend in the xor, and all $2^n - 1$ values other than M_j are equally likely for it, whence Checksum can assume $2^n - 1$ values, each equiprobable, and Final can assume $2^n - 1$ values, each equiprobable. We are in the setting of game G and the adversary's chance to forge is at most $2^{n-\tau}/(2^n - 1)$.

Case 6: $A = A'$, $c = m$, M' and C partial. Because $C \neq C'$ either there is a smallest j where $C'_j \neq C_j$ or else $C'_1 \dots C'_c = C_1 \dots C_c$ but $C'_* \neq C_*$. For the former case, proceed like Case 5. For the latter case the adversary may be able to compute $\text{Checksum}' = M'_1 \oplus \dots M'_c \oplus M'_* 10^*$ but also $\text{Checksum} = M'_1 \oplus \dots M'_c \oplus M_* 10^*$. These two values are necessarily distinct, due to the 10^* padding. Even if the adversary is given both and knows Final' the image of Checksum under $\pi^{Nc}\$$ will be a uniformly random among the $2^n - 1$ points that are not Final' . We are again left in the situation of game G , and the adversary's probability to forge is at most $2^{n-\tau}/(2^n - 1)$. This completes the proof. $\square$

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

Fig. 9. Stretch-then-shift hash. Largest $\mathcal{X} = [0..\text{domSize}(c) - 1]$ $H_K^c(x) = (\text{Stretch}(K) \ll x)[1..128]$ is strongly xor-universal when $c \in [1..20]$, $K \in \{0, 1\}^{128}$, $x \in \mathcal{X}$, and $\text{Stretch}(K) = K \parallel (K \oplus (K \ll c))$.

7.2. Stretch-then-Shift Hash

OCB employs a new hash function, “Init” in Fig. 3, to map the nonce N to the initial offset Δ . We aimed to construct a hash that would need at most one AES computation no matter what N -values are presented, but would usually reuse a previously computed AES output, plus a couple of shifts and xors, if N is one more than it was last time around. Our method applies AES to the top 122 bits of N to create an initial value K_{top} , but then modifies K_{top} in a simple way using the lower six bits of N , call them Bottom. Specifically, we return the first 128 bits of $H_{K_{\text{top}}}(\text{Bottom}) = (K_{\text{top}} \parallel (K_{\text{top}} \oplus (K_{\text{top}} \ll 8))) \ll \text{Bottom}$. There is no deep intuition motivating the formula; it is just the simplest method we found that we could prove to work. We start with the needed definitions.

DEFINITION. Let $\mathcal{K}$ be a finite set and let $H: \mathcal{K} \times \mathcal{X} \rightarrow \{0, 1\}^n$ 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* and the second we call *universal-1*.

THE TECHNIQUE. We aim to construct strongly xor-universal hash-functions $H: \mathcal{K} \times \mathcal{X} \rightarrow \{0, 1\}^n$ where $\mathcal{K} = \{0, 1\}^{128}$, $\mathcal{X} = [0..\text{domSize} - 1]$, and $n = 128$. We want domSize to be at least some modest-size number, say $\text{domSize} \geq 64$, and intend that computing $H_K(x)$ be extremely fast. Computation of H should not require large tables, preprocessing of K , or special hardware support.

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

How to stretch K ? It seems natural to have $\text{stretch}(K)$ begin with K , so let us assume that $\text{stretch}(K) = K \parallel s(K)$ for some function s . It is easy to see that $s(K) = K$ and $s(K) \ll c$ will not work, but $s(K) = K \oplus (K \ll c)$, for some constant c , looks plausible. We now demonstrate that, for well-chosen c , this function does the job.

ANALYSIS. To review, we are considering the hash functions $H_K^c(x) = (\text{Stretch} \ll x)[1..128]$ where $\text{Stretch} = \text{stretch}(K) = K \parallel (K \oplus (K \ll c))$ and $c \in [0..127]$. We would like to know the maximal value of domSize for which $H_K(x)$ is xor-universal on the domain $\mathcal{X} = [0..\text{domSize}(c) - 1]$. 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 only A ’s rows i to $i + 127$. Then $H_K^c(x) = A_{x+1} K$, the product in $\text{GF}(2)$ 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 $\text{GF}(2)$. We would like to ensure that, for arbitrary $0 \leq i < j < \text{domSize}(c)$ 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 $\text{GF}(2)$ for all $1 \leq i < j \leq \text{domSize}(c)$. Thus $\text{domSize}(c)$ can be computed as the largest number $\text{domSize}(j)$ such that $B_{i,j}$ is full rank, over $\text{GF}(2)$, for all $1 \leq i < j \leq \text{domSize}(j)$. Recalling the universal-1 property we also demand that A_i have full rank for all $1 \leq i \leq \text{domSize}(c)$. Now for any c , the number of matrices $A_{i,j}$ to consider is at most 2^{13} , and finding the rank in $\text{GF}(2)$ of that many 128×128 matrices is not a difficult calculation.

Our results are tabulated in Fig. 9. 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 OCB and single out the following result:

Lemma 3. *Let $H: \{0, 1\}^{128} \times [0..63] \rightarrow \{0, 1\}^{128}$ be defined by $H_K(x) = (\text{Stretch} \ll x)$ $[1..128]$ where $\text{Stretch} = K \parallel (K \oplus (K \ll 8))$. Then, H is strongly xor-universal.*

EFFICIENCY. On 64-bit computers, assuming $K \parallel (K \oplus (K \ll 8))$ is precomputed and in memory, the value of $H_K(x)$ can be computed by three memory loads and two multi-precision 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.

7.3. Instantiating the TBC

We show that $\tilde{E} = \text{Tw}[E, \tau]$ is a good TBC if E is a good blockcipher. In formalizing this, bidirectional tweaks are those of the form (N, i) . Our result is again, for now, in the information-theoretic setting.

Lemma 4. *Fix $n = 128$ and $\tau \in [0..n]$ and $E = \mathbb{E}^n$. Let $\tilde{E} = \text{Tw}[E, \tau]$ and $\mathcal{B} = \mathcal{N} \times \mathbb{N}_1$. Let $\mathcal{A}$ be an adversary that asks at most q queries, all with i -value less than 2^{120} . Then $\text{Adv}_{\tilde{E}}^{\mathcal{B}\text{-PTP}}(\mathcal{A}) \leq 6q^2/2^n$.*

Proof. We generalize the adversary's capabilities in attacking $\text{Tw}[E, \tau]$; see Fig. 10 for the construction we will call TW. There we write π instead $\tilde{E}_K$. The adversary, which we still refer to as $\mathcal{A}$, may now ask queries we will 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 $(1, W, \lambda)$, $(2, W, \text{Top}, \text{Bottom}, \lambda)$, $(3a, W, \text{Top}, \text{Bottom}, \lambda)$, or $(3b, Z, \text{Top}, \text{Bottom}, \lambda)$. 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 is 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.

Fig. 10. 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 4 hinges on establishing the rarity of nontrivial collisions among the inputs or outputs of π .

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 Fig. 5. 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 proven secure in Lemma 3. 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 $\text{Tw}[E, \tau]$ are also represented in the abilities of an adversary attacking $\text{TW}[E, \tau]$ 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 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 $\text{GF}(2^n)$. 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 $\tilde{\pi}$ (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—e.g., the 20th internal query colliding with the 6th—can be summarized by writing something like $\text{Coll}(3a, 1)$, interpreted as saying that first there was a type-3a internal query $(W, \text{Top}, \text{Bottom}, \lambda)$, 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 $W + \text{Initial} + \lambda L = W' + \lambda' L$, and Initial is uniform and independent of all other values named in the formula: we select the type-B output K_{top} of π uniformly at random, and H is universal-1, making $H_{K_{\text{top}}}(\text{Bottom})$ 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.

- $\Pr[\text{Coll}(A, B)] = \Pr[\text{Coll}(B, A)] = 0$ since the adversary is not allowed to query with $\text{Top} = 0$.
- $\Pr[\text{Coll}(3a, 3a)] = \Pr[W + \text{Initial} + \lambda L = W' + \text{Initial}' + \lambda' L]$. Queries may not be repeated, so $(W, \text{Top}, \text{Bottom}, \lambda) \neq (W', \text{Top}', \text{Bottom}', \lambda')$. Suppose $\text{Top} \neq \text{Top}'$. Then K_{top} and $K_{\text{top}'}$ are uniform and independent, making Initial and Initial' uniform and independent by the universal-1 property of H , so the probability in question is at most 2^{-n} . Suppose $\text{Top} = \text{Top}'$ but $\text{Bottom} \neq \text{Bottom}'$. Then $K_{\text{top}} = K_{\text{top}'}$ 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 $\text{Top} = \text{Top}'$ and $\text{Bottom} = \text{Bottom}'$ and $\lambda = \lambda'$ then we know that $W \neq W'$ and the probability of collision is 0. Finally, if $\text{Top} = \text{Top}'$ and $\text{Bottom} = \text{Bottom}'$ 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} .
- $\Pr[\text{Coll}(2, 3b)] = \Pr[Y = \text{Initial}' + \lambda' L]$. This is at most 2^{-n} as, for example, Initial' is uniform and independent of Y, λ' , and L .
- $\Pr[\text{Coll}(3b, 2)] = \Pr[W + \text{Initial} + \lambda L = W' + \text{Initial}' + \lambda' L]$. Since λ -values must be distinct between TYPE-2 and TYPE-3b the probability is $\Pr[cL = W + \text{Initial} + W' + \text{Initial}']$ 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. $\square$

7.4. Wrapping Up

The following concretizes a claim, by now folklore, from Rogaway and Shrimpton [41, Section 7]. It is one half of the equivalence between ae-security and priv+auth security: an AE scheme that is secure in the priv and auth senses is secure under the all-in-one definition, too. We include a proof, as we don't know where else in the literature one appears.

Lemma 5. *Let Π be an AE scheme. Let $\mathcal{A}$ be an adversary that asks at most q_d decryption queries. Then there are adversaries $\mathcal{A}_{\text{priv}}$ and $\mathcal{A}_{\text{auth}}$ such that*

$$\mathbf{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) \leq \mathbf{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}_{\text{priv}}) + q_d \cdot \mathbf{Adv}_{\Pi}^{\text{auth}}(\mathcal{A}_{\text{auth}}).$$

Adversaries $\mathcal{A}_{\text{priv}}$ and $\mathcal{A}_{\text{auth}}$ are explicitly constructed from $\mathcal{A}$ in the proof of this lemma. They have time and query complexity similar to that of $\mathcal{A}$.

Proof. Adversary $\mathcal{A}_{\text{priv}}$ is constructed as follows. It runs adversary $\mathcal{A}$. When $\mathcal{A}$ makes a left-oracle query of (N, A, M) adversary $\mathcal{A}_{\text{priv}}$ asks its own oracle (N, A, M) and gets a response C , which it returns to $\mathcal{A}$. When $\mathcal{A}$ makes a right-oracle query of (N, A, C) , it returns $\perp$ to $\mathcal{A}$. When $\mathcal{A}$ halts with an output of b , adversary $\mathcal{A}_{\text{priv}}$ outputs the same bit b and halts.

Adversary $\mathcal{A}_{\text{auth}}$ is constructed as follows. It selects a random $j \leftarrow [1..q_d]$. It then runs adversary $\mathcal{A}$. When $\mathcal{A}$ makes a left-oracle query of (N, A, M) adversary $\mathcal{A}_{\text{auth}}$ asks its own oracle (N, A, M) and gets a response C , which it returns to $\mathcal{A}$. When $\mathcal{A}$ makes its i th right-oracle query of (N, A, C) , adversary $\mathcal{A}_{\text{auth}}$ returns $\perp$ if $i \neq j$, while, if $i = j$, it outputs (N, A, C) (as its attempted forgery) and then halts.

Omitting oracle-argument placeholders $(\cdot, \cdot, \cdot)$ for improved readability, we have that

$$\begin{aligned} \mathbf{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) &= \Pr[\mathcal{A}^{\mathcal{E}_K, \mathcal{D}_K} \Rightarrow 1] - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1] \\ &= (\Pr[\mathcal{A}^{\mathcal{E}_K, \mathcal{D}_K} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathcal{E}_K, \perp} \Rightarrow 1]) \\ &\quad + (\Pr[\mathcal{A}^{\mathcal{E}_K, \perp} \Rightarrow 1] - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1]) \\ &\leq \Pr[\text{SomeForge}] + \mathbf{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}_{\text{priv}}) \\ &\leq \sum_{i=1}^{q_d} \Pr[\text{FirstForge}_i] + \mathbf{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}_{\text{priv}}) \\ &\leq q_d \cdot \Pr[\text{FirstForge}] + \mathbf{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}_{\text{priv}}) \end{aligned}$$

where **SomeForge** is the event that adversary $\mathcal{A}$, in the course of its execution with oracles $\mathcal{E}_K, \mathcal{D}_K$, asks its decryption oracle *some* query that results in a non- $\perp$ return value; where **FirstForge** _{i} is the event that $\mathcal{A}$, in the course of its execution with oracles $\mathcal{E}_K, \mathcal{D}_K$, gets its *first* non- $\perp$ oracle response on query- i ; and where **FirstForge** is the

event that $\mathcal{A}$, in the course of its execution with oracles $\mathcal{E}_K, \mathcal{D}_K$, gets its first non- $\perp$ return value in response to its j th query, for j uniform in $[1..q_d]$. Our construction of $\mathcal{A}_{\text{auth}}$ ensures that a decryption query by $\mathcal{A}$ that gets a first non- $\perp$ response always corresponds to a successful forgery by the adversary $\mathcal{A}_{\text{auth}}$ we have constructed; that is, $\text{Adv}_{\Pi}^{\text{auth}}(\mathcal{A}_{\text{auth}}) = \Pr[\mathcal{A}_{\text{auth}}^{\mathcal{E}_K} \text{ forges}] \geq \Pr[\text{FirstForge}]$. This completes the proof. $\square$

We now have all the needed elements to conclude the security of OCB. First, combining Lemmas 2 and 5 immediately establishes the ae-security of ΘCB based on an idealized TBC.

Lemma 6. Fix $n = 128$ and $\tau \in [0..n]$ and $\tilde{\mathbb{E}} = \tilde{\mathbb{E}}^{\mathcal{T}_{\Theta\text{CB}}, n}$. Let $\Pi = \Theta\text{CB}[\tilde{\mathbb{E}}, \tau]$. Let $\mathcal{A}$ be an adversary that asks at most q_d decryption queries. Then $\text{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) \leq 1.001 q_d/2^{\tau}$.

Next we replace the idealized TBC of $\Theta\text{CB}[\tilde{\mathbb{E}}, \tau]$ by the TBC one gets from the Tw construction applied to an idealized (but not tweakable) blockcipher. Combining Lemmas 4 and 6 then gives the following. In the theorem statement, the number of blocks asked by an adversary refers to the total number of n -bit plaintext or ciphertext blocks, any fractional final blocks counting as one block, plus the number of associated-data blocks, any fractional final blocks again counting as one block.

Lemma 7. Fix $n = 128$ and $\tau \in [0..n]$ and $\mathbb{E} = \mathbb{E}^n$. Let $\Pi = \text{OCB}[\mathbb{E}, \tau]$. Let $\mathcal{A}$ be an adversary that asks at most q queries, at most q_d of them decryption queries, and at most σ blocks. Then $\text{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) \leq 6(\sigma + 2q)^2/2^n + 1.1 q_d/2^{\tau}$.

Proof. We construct an adversary $\mathcal{D}$ as follows. It has an oracle for a TBC taking tweaks $T \in \mathcal{T}_{\Theta\text{CB}}$ and inputs $X \in \{0, 1\}^n$; and another oracle for the inverse of this TBC, taking tweaks $T \in \mathcal{T}_{\Theta\text{CB}}$ and inputs $Y \in \{0, 1\}^n$. Adversary $\mathcal{D}$ runs adversary $\mathcal{A}$. When $\mathcal{A}$ makes an encryption query of (N, A, M) adversary $\mathcal{D}$ services this query by encrypting M with respect to N and A using the ΘCB algorithm but employing its first oracle in lieu of all forward-direction blockcipher calls. When $\mathcal{A}$ makes a decryption query of (N, A, C) adversary $\mathcal{D}$ services it by decrypting C with respect to N and A using the ΘCB algorithm but employing its first oracle for forward-direction blockcipher calls and its second oracle for inverse-direction blockcipher calls. Note that all calls made by $\mathcal{D}$ use tweaks from $\mathcal{T}_{\Theta\text{CB}}$ and that inverse calls will only use tweaks of the form (N, i) . We may assume that all calls employ i -values of less than 2^{64} , for, otherwise, $\sigma \geq 2^{64}$ and the result is vacuous. The total number of queries asked by $\mathcal{D}$ will be at most $\sigma + 2q$, the $2q$ accounting for the fact that there is one TBC call for the Checksum and one TBC call for the processing of the nonce to produce the initial offset. Now

$$\begin{aligned}
 \text{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) &= \Pr[\mathcal{A}^{\text{OCB}[\mathbb{E}, \tau].\mathcal{E}, \text{OCB}[\mathbb{E}, \tau].\mathcal{D}} \Rightarrow 1] - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1] \\
 &= (\Pr[\mathcal{A}^{\text{OCB}[\mathbb{E}, \tau].\mathcal{E}, \text{OCB}[\mathbb{E}, \tau].\mathcal{D}} \Rightarrow 1] - \Pr[\mathcal{A}^{\Theta\text{CB}[\tilde{\mathbb{E}}, \tau].\mathcal{E}, \Theta\text{CB}[\tilde{\mathbb{E}}, \tau].\mathcal{D}} \Rightarrow 1]) + \\
 &\quad (\Pr[\mathcal{A}^{\Theta\text{CB}[\tilde{\mathbb{E}}, \tau].\mathcal{E}, \Theta\text{CB}[\tilde{\mathbb{E}}, \tau].\mathcal{D}} \Rightarrow 1]) - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1]) \\
 &= \text{Adv}_{\text{Tw}[\mathbb{E}, \tau]}^{\mathcal{B}\text{-prp}}(\mathcal{D}) + \text{Adv}_{\Theta\text{CB}[\tilde{\mathbb{E}}, \tau]}^{\text{ae}}(\mathcal{A}) \\
 &\leq 6(\sigma + 2q)^2/2^n + 1.1 q_d/2^{\tau}.
 \end{aligned}$$

The first term is by Lemma 4 and the observations made previously concerning adversary $\mathcal{D}$'s attack. The second term is by Lemma 6. $\square$

Finally we can quantify the security of OCB when instantiated with a “real” blockcipher E .

Theorem 1. *Fix $n = 128$ and $\tau \in [0..n]$ and a blockcipher $E: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$. Let $\Pi = \text{OCB}[E, \tau]$. Let $\mathcal{A}$ be an adversary that asks at most q queries, at most q_d of them decryption queries, and at most σ blocks. Then there is an adversary $\mathcal{A}_{\text{prp}}$, explicitly constructed by the proof of this theorem, for which*

$$\mathbf{Adv}_E^{\pm\text{prp}}(\mathcal{A}_{\text{prp}}) \geq \mathbf{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) - 6(\sigma + 2q)^2/2^n - 1.1 q_d/2^\tau.$$

Its time complexity is similar to that of $\mathcal{A}$, and it asks at most $\sigma + 2q$ queries.

Proof. Adversary $\mathcal{A}_{\text{prp}}$ is constructed as follows. It runs $\mathcal{A}$. When $\mathcal{A}$ asks an encryption query of (N, A, M) adversary $\mathcal{A}_{\text{prp}}$ services this query by using the OCB construction with tag length τ , but employing its own left oracle in lieu of each forward blockcipher computation. When $\mathcal{A}$ asks a decryption query of (N, A, C) adversary $\mathcal{A}_{\text{prp}}$ services this query by using the OCB construction with tag length τ , but employing its own left oracle in lieu of each forward blockcipher computation and employing its right oracle in lieu of each inverse blockcipher computation. Now

$$\begin{aligned} \mathbf{Adv}_{\Pi}^{\text{ae}}(\mathcal{A}) &= \Pr[\mathcal{A}^{\text{OCB}[E, \tau].\mathcal{E}, \text{OCB}[E, \tau].\mathcal{D}} \Rightarrow 1] - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1] \\ &= (\Pr[\mathcal{A}^{\text{OCB}[E, \tau].\mathcal{E}, \text{OCB}[E, \tau].\mathcal{D}} \Rightarrow 1] - \Pr[\mathcal{A}^{\text{OCB}[\mathbb{E}, \tau].\mathcal{E}, \text{OCB}[\mathbb{E}, \tau].\mathcal{D}} \Rightarrow 1]) + \\ &\quad (\Pr[\mathcal{A}^{\text{OCB}[\mathbb{E}, \tau].\mathcal{E}, \text{OCB}[\mathbb{E}, \tau].\mathcal{D}} \Rightarrow 1] - \Pr[\mathcal{A}^{\$, \perp} \Rightarrow 1]) \\ &\leq \mathbf{Adv}_E^{\pm\text{prp}}(\mathcal{A}_{\text{prp}}) + 6(\sigma + q)^2/2^n + 2q_d/2^\tau \end{aligned}$$

by the definition of $\mathcal{A}_{\text{prp}}$ and Lemma 7. The result follows. $\square$

8. Conclusions

LIMITATIONS. There are three limitations on OCB that we would like to emphasize. The first is that OCB's security crucially depends on the encrypting party not repeating a nonce. The mode should never be used in situations where that can't be assured; one should instead employ a misuse-resistant AE scheme [41]. These include AES-GCM-SIV [13, 14], COLM [2], and Deoxys-II [23]. A second limitation of OCB is its birthday-bound degradation in provable security. This limitation implies that, given OCB's 128-bit block size, one must avoid operating on anything near 2^{64} blocks of data. The RFC on OCB [27] asserts that a given key should be used to encrypt at most 2^{48} blocks (4 petabytes), including the associated data. Practical AE modes that avoid the birthday-bound degradation in security are now known [14, 19, 20, 23, 40]. Finally, OCB uses both the forward and the backward direction of its underlying blockcipher.

Minematsu has devised an elegant approach for avoiding use of a blockcipher's inverse in an OCB-like construction [33].

THE OCB2 BUG. We mentioned in the Introduction the devastating bug on OCB2 discovered by Inoue, Iwata, Minematsu, and Poettering [17]. What was the root cause of this bug [17], and why does OCB3 escape it? We offer two high-level explanations.

One is to claim that OCB2—and OCB1 and OCB3 as well—are over-optimized. In particular, the constructions twice omit post-whitening xors [38], switching from the XEX construction to the XE construction [30, 38] for enciphering the Checksum and for Vernam-style encrypting the final block of plaintext. (OCB2 and OCB3 also use XE for processing the AD.) Shaving off these two xors increased conceptual complexity and proof complexity for an insignificant improvement in speed.

But the intermixing of XE and XEX constructions is not by itself a problem; what made this turn out so badly for OCB2 was the muddy (and therefore error-prone) abstraction boundary for dealing with TBCs that combine XE and XEX [38, Section 8]. Rather than partitioning the tweak space into unidirectional and bidirectional tweaks, as we did with OCB3, a one-bit tag was appended to each tweak—but then the tag was not treated as an intrinsic part of the tweak. This failed to properly capture the intent that things like π_4^N (with a bold π) and $\bar{\pi}_4^N$ (with a bar over the π) and π_4^N in [38, Fig. 1] all need to be instantiated so as to approximate independent permutations. Properly formalizing a partitioned-tweakspace PRP helped steer us clear of the error to which OCB2 succumbed.

ISO STANDARDIZATION. The OCB2 bug was made more significant because it had been standardized in ISO/IEC 19772 [18]. Unfortunately, nobody on that standardization committee had told us that they were planning to standardize this scheme. Had they, we would have explained that we had already abandoned OCB2 in favor of a cleaner and more empirically informed redesign, OCB3, an unpublished draft of which already existed. The same ISO standard included another faulty AE scheme: a badly misspecified version of encrypt-then-MAC [34]. These two problems suggest, to us, that a rather insular standardization process was then in place.

PATENTS. At present, OCB is not widely used. This is largely due to there having been potentially relevant patents held by multiple parties. This was the reason that OCB1 fell out of being a mandatory mechanism of 802.11 [16], and the reason it was not subsequently selected for a NIST Special Publication. The limited adoption of OCB exemplifies the often poisonous impact of patents on cryptographic practice.

The second author, who held the patents most directly reading against OCB, has renounced all OCB-relevant IP; all OCB-relevant patents of his have been placed in the public domain. While Rogaway had previously licensed all open-source software and all software for non-military use, IP concerns did not go away.

FINAL REMARK. As explained in Sect. 5, the initial reason for developing OCB1 was to improve the efficiency of authenticated encryption. But it soon became clear to us that AE schemes had a second benefit: that they could improve *usability*. By moving closer to what an ideal encryption scheme would deliver (and, also, by adjusting an encryption-scheme's syntax to include AD and avoid per-message randomness) one could make symmetric encryption less prone to misuse and less reliant on fussy compo-

sition techniques [4,34]. OCB, and AE more broadly, helped birth a reconceptualization of symmetric encryption. One abandons weak security notions and schemes that are hard to correctly use and moves toward strong security notions and schemes that are easier to use well. This, not speed, may be the lasting legacy of AE.

Acknowledgements

The authors thank all who were involved in the CAESAR effort, starting with its organizer, Dan Bernstein, and fellow submitters. We thank Tetsu Iwata for his work in organizing this JoC special issue, and the referees who returned helpful comments on our manuscript. Thanks to the NSF for many years of support. NSF 0904380 was used for the development of OCB3, NSF 1314885 funded renewed work for the CAESAR submission, and NSF 1717542 funded the preparation of this manuscript.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- [1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology* 15(2), pp. 103–127, 2002
- [2] E. Andreeva, A. Bogdanov, N. Datta, A. Luykx, B. Mennink, M. Nandi, E. Tischhauser, K. Yasuda. COLM v1. September 15, 2016. CAESAR submission. <https://competitions.cr.yp.to/>
- [3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: analysis of the DES modes of operation. *Proceedings of 38th Annual Symposium on Foundations of Computer Science (FOCS 97)*, IEEE, 1997
- [4] M. Bellare, C. Namprempe. Authenticated encryption: relations among notions and analysis of the generic composition paradigm. *J. Cryptol.*, 21(4), pp. 469–491, 2008
- [5] D. Bernstein. The Poly1305-AES message-authentication code, in *FSE 2005*, LNCS vol. 3557 (Springer, New York, 2005), pp. 32–49
- [6] D. Bernstein, D. Bernstein. ChaCha, a variant of Salsa20. <https://cr.yp.to/chacha/chacha-20080128.pdf>. Manuscript, 2008
- [7] D. Bernstein. CAESAR competition webpage at <https://competitions.cr.yp.to/caesar-submissions.html> Last visited 2020.06.19
- [8] Cryptographic Engineering Research Group (CERG). Hardware Benchmarking of CAESAR Candidates. Webpage <https://cryptography.gmu.edu/athena/index.php?id=CAESAR>. Last visited 2019.12.19
- [9] M. Dworkin. Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality. NIST Special Publication 800-38C. May 2004
- [10] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. November 2007
- [11] V. Gligor, P. Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes, in *FSE 2001*, LNCS vol. 2355, (Springer, New York, 2001), pp. 92–108

- [12] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, vol. 28, April 1984, pp. 270–299
- [13] S. Gueron, A. Langley, Y. Lindell: AES-GCM-SIV: Nonce misuse-resistant authenticated encryption. IETF RFC 8452. April 2019
- [14] S. Gueron, Y. Lindell: GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* 2015
- [15] E. Homsirikamol, K. Gaj. Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: the CAESAR contest case study. *International Conference on Field Programmable Technology (ICFPT 2017)*. IEEE Press, pp. 120–127, 2017
- [16] IEEE Standard 802.11i-2004. Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Medium Access Control (MAC) Security Enhancements. 2004
- [17] A. Inoue, T. Iwata, K. Minematsu, B. Poettering. Cryptanalysis of OCB2: attacks on authenticity and confidentiality, in *Advances in Cryptology—CRYPTO 2019* (Springer, New York, 2019)
- [18] ISO/IEC 19772. Information technology-Security techniques-Authenticated encryption. First edition, 2009-02-15. OCB scheme within standard subsequently deprecated
- [19] T. Iwata. Authenticated encryption mode for beyond the birthday bound security. *AFRICACRYPT 2008*, LNCS vol. 5023, Springer, pp. 125–142, 2008
- [20] T. Iwata. New blockcipher modes of operation with beyond the birthday bound security. *FSE 2006*, LNCS 4047, pp. 310–327, 2006
- [21] T. Iwata, K. Yasuda. BTM: a single-key, inverse-cipher-free mode for deterministic authenticated encryption. *SAC 2009*, LNCS vol. 5667, Springer, pp. 313–330, 2009
- [22] T. Iwata, K. Yasuda. HBS: a single-key mode of operation for deterministic authenticated encryption. *FSE 2009*, LNCS vol. 5665, Springer, 2009, pp. 394–415
- [23] J. Jean, I. Nikolić, T. Peyrin, Y. Seurin: Deoxys v1.41. October 12, 2016. CAESAR submission. <https://competitions.cr.yp.to/>
- [24] C. Jutla, Encryption modes with almost free message integrity. *EUROCRYPT 2001*, LNCS vol. 2045, Springer, pp. 529–544, 2001
- [25] T. Krovetz. OCB for block ciphers without 128-bit blocks. Internet Draft draft-krovetz-ocb-wideblock-00. May 11, 2018
- [26] T. Krovetz, P. Rogaway. The software performance of authenticated-encryption modes. *FSE 2011*, LNCS vol. 6733, Springer, pp. 306–327, 2011
- [27] T. Krovetz, P. Rogaway. The OCB authenticated-encryption algorithm. IETF RFC 7253. May 29, 2014
- [28] J. Katz, M. Yung. Unforgeable encryption and adaptively secure modes of operation. *FSE 2000*, LNCS vol. 1978, Springer, 2001
- [29] T. Krovetz, P. Rogaway. The OCB authenticated-encryption algorithm. Internet Draft draft-krovetz-ocb-00.txt. March 2005
- [30] M. Liskov, R. Rivest, D. Wagner. Tweakable block ciphers. *CRYPTO 2002*, LNCS vol. 2442, Springer, pp. 31–46, 2002
- [31] D. McGrew, 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] Y. Nir, A. Langley. ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 8439. May 2015
- [33] K. Minematsu: Parallelizable rate-1 authenticated encryption from pseudorandom functions. *EUROCRYPT 2014*, LNCS vol. 8441, Springer, pp. 275–292, 2014
- [34] C. Namprempre, P. Rogaway, T. Shrimpton. Reconsidering generic composition. *EUROCRYPT 2014*. LNCS vol. 8441, Springer, pp. 257–274, 2014
- [35] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446. August 2018
- [36] R. Reyhanitabar, S. Vaudenay, and D. Vizár. Authenticated encryption with variable stretch. *ASIACRYPT* (1), pp. 396–425, 2016
- [37] P. Rogaway. Authenticated-encryption with associated-data. *CCS 2002*, ACM Press, 2002
- [38] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. *ASIACRYPT 2004*, LNCS vol. 3329, Springer, pp. 16–31, 2004

- [39] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transaction on Information and System Security*, 6(3), pp. 365–403, 2003. Earlier version, with T. Krovetz, in *CCS 2001*
- [40] T. Peyrin, Y. Seurin: Counter-in-Tweak: Authenticated encryption modes for tweakable block ciphers. *CRYPTO 2016*, LNCS vol. 9814, Springer, pp. 33–63, 2016
- [41] P. Rogaway, T. Shrimpton. A provable-security treatment of the key-wrap problem. *EUROCRYPT 2006*, LNCS vol. 4004, Springer, pp. 373–390, 2006
- [42] SUPERCOP website. Toolkit and website from VAMPIRE Lab. <https://bench.cr.yp.to/supercop.html>. Last visited 2019.12.08
- [43] D. Whiting, R. Housley, N. Ferguson. AES encryption & authentication using CTR mode & CBC-MAC. IEEE P802.11 doc 02/001r2, May 2002
- [44] H. Wu, B. Preneel. AEGIS: a fast authenticated encryption algorithm. SAC 2013, LNCS vol. 8282, pp. 185–201. Also CAESAR submission document <https://competitions.cr.yp.to/round3/aegisv11.pdf>. 2016

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.