Parallel Verification of Serial MAC and AE Modes*

Kazuhiko Minematsu[1], Akiko Inoue[1],
Katsuya Moriwaki[2], Maki Shigeri[2], and Hiroyasu Kubo[2]

[1] NEC, Kawasaki Japan
k-minematsu@nec.com, a_inoue@nec.com
[2] NEC Solution Innovators, Hokuriku, Japan
m-shigeri_pb@nec.com,h-kubo@nec.com

Keywords: Mode of Operation · MAC · Authenticated Encryption · Parallel verification · CMAC · CCM · Romulus

Abstract. A large number of the symmetric-key mode of operations, such as classical CBC-MAC, have serial structures. While a serial mode gives an implementation advantage in terms of required memory or footprint compared to the parallel counterparts, it wastes the capability of parallel process even when it is available. The problem is becoming more relevant as lightweight cryptography is going to be deployed in the real world. In this article, we propose an alternative implementation strategy for serial MAC modes and serial authenticated encryption (AE) modes that allows 2-block parallel operation for verification/decryption. Our proposal maintains the original functionality and security. It is simple yet novel, and generally applicable to a wide range of existing modes including two NIST recommendations, CMAC and CCM. We demonstrate the effectiveness of our proposal by showing several case studies with software implementations.

1 Introduction

Lightweight cryptography is a subfield of symmetric-key cryptography that aims at designing cryptographic functions that perform well under constrained environments where standards might not perform well or not be implementable at all. One of the major directions is lightweight cryptographic primitives, such as block ciphers [10, 19, 21], tweakable block ciphers [9, 12], and cryptographic permutations [15, 25, 28]. As well as the primitives, the modes for encryption, authentication or authenticated encryption (AE) have been actively studied from the viewpoint of lightweight cryptography.

The hardware implementation size is an important metric for lightweight modes. When one wants to design a mode with small implementation size, the *state memory* (memory for keeping the internal chaining value or precomputed

* This is an extended version of a paper presented at SAC 2021.

secret values) can occupy a significant amount in the total size compared to the core cryptographic logics. Parallelizable modes allow to process input blocks in parallel, however this imposes some additional state memory blocks, which is disadvantageous in terms of size. From this reason, making the mode entirely serial is quite effective and deemed as a common strategy for lightweight modes. This is illustrated by the ongoing NIST Lightweight Cryptography project for standardizing lightweight authenticated encryption (hereafter NIST LWC) [1], where a large fraction of proposals are serial. However, losing parallelizability means losing performance gain even when parallel processing is available. This can occur in practice, *e.g.*, when lots of tiny sensor devices are connected to the edge server – which is typically much more powerful than sensors – that aggregates information from the sensors. This exhibits a dilemma for lightweight cryptographic schemes.

In this paper, we propose a new simple implementation strategy for serial modes. More specifically we consider serial MAC (authentication) modes and AE modes, where each input block must be serially processed in the original specifications. Our strategy allows to process two input blocks in parallel in the verification of MAC modes and the (authenticated) decryption of AE modes. The idea, which we call *pincer verification*, is to process the input message from the top and the bottom, and performs the verification check at the middle of the message. It is quite intuitive and simple, yet to our knowledge never studied in the literature. The easiest example is the plain CBC-MAC with message $M = (M[1], \ldots, M[m])$ and tag T, where $|M[i]| = |T| = n$. The pincer verification performs the regular CBC-MAC computation from $M[1], M[2], \ldots$, using the block cipher encryption E_K. At the same time, it performs the “backward” computation from T, $M[m], M[m-1], \ldots$, using the block cipher decryption E_K^{-1}, and verifies at the middle. The idea may look almost trivial, however, to deal with more complex modes of operations, we need an abstract model to clarify when and how such implementation is possible. In addition we extend the idea to AE modes. When we emphasize that the target is AE we may call it *pincer decryption* [3]. For both MAC and AE modes, the functional equivalence needs to be maintained. We clarify the condition that enables pincer verification that maintains the original functionality. More precisely, the result of pincer verification (*i.e.*, a binary verification result for a MAC mode, and a decrypted plaintext for an AE mode when verification is successful) is always the same as the original. Thus, we can use pincer verification without any security loss. We look into the current MAC and AE modes, and show that pincer verification is applicable to a wide variety of them. In particular it is applicable to the most of known CBC-MAC based MAC modes, including the NIST recommendation, CMAC [3]. Regarding AE, pincer verification is applicable to some of the state-of-the-art serial AE modes including some submissions to NIST LWC [1]. Finally, although the structure is not fully serial, the NIST recommendation CCM [2] also allows a variant of pincer decryption (see below). Parallelizability of our proposal is

[3] We use the term pincer verification to cover both MAC and AE modes. Note that pincer verification for AE is in fact authenticated decryption, not just verification.

limited, still it doubles the verification performance in theory. Considering that no further fundamental speedup is not possible for single message, our proposal pushes the performance limit of serial modes and can play a meaningful role in practice.

We show three case studies to demonstrate the utility of pincer verification. Pincer verification is generally useful for software on multi-core processors. That is, one core for the forward direction starting at the first message block and the other for the backward direction starting at the last message block and the tag. We basically assume that the verification routine takes the entire input kept in memory. Although it is a limitation, this assumption holds for the typical API of cryptographic library.

Our first case study follows the aforementioned basic implementation scenario. The target algorithm is Romulus [27, 29], a finalist of the NIST LWC. The target platform is a low-end 32-bit dual-core microcontroller ESP32[4], which is one of the common choices for IoT applications, for its built-in Wifi and Bluetooth. We show that even on this constrained device the pincer decryption can double the performance for a reasonable length of message. The advantage of ESP32's dual-core operation was explored in the post-quantum cryptography [45], however to our knowledge no concrete application has been shown in the symmetric-key literature.

Our second case study takes CMAC[5], which is the standard MAC mode recommended by NIST. On x86 platforms equipped with AES instruction set (AES-NI), the pincer verification doubles the performance utilizing the pipeline as in the same manner to the parallelizable (*e.g.*, CTR) modes. An important remark is that our code runs on *single core*. It runs about 1.65 cycles per byte for 1024-byte message where the regular implementation runs at 3.35. Thus, we double the verification performance of CMAC per core from the straightforward (but previously the fastest) implementation.

Our third case study is CCM, a NIST-recommended AE mode based on AES. It has been quite widely deployed, such as TLS, IPSec, Wifi (WPA2) and so on. CCM composes CBC-MAC and CTR mode in a similar way to the MAC-then-encrypt generic composition. As mentioned earlier, CCM does not perfectly fall into our framework since it is not fully serial, however it still benefits from pincer verification technique due to its complete serial MAC part (CBC-MAC). We show that, on x86 platforms with AES-NI, a variant of pincer verification can halve the computation cost of decryption from the previous fastest implementation strategy (*e.g.*, OpenSSL [6]) that interleaves CBC-MAC and CTR decryption. Our concrete implementation achieves 1.78 cycles per byte for CCM decryption. It is faster than the previous one by a factor of two. See Sections 6,7 and 8 for more details of these case studies.

[4] `https://www.espressif.com/en/products/socs/esp32`

[5] CMAC is a generic mode for 128-bit block ciphers. This paper assumes AES-128 as the underlying block cipher for CMAC. The same applies to CCM.

[6] `https://www.openssl.org/`

Organization. This paper is organized as follows. After the introduction at Section 1, Section 2 describes some technical backgrounds. Section 3 describes the basic idea of pincer verification for MAC modes, and Section 4 extends it to AE modes. Section 5 shows the applicability of pincer verification to existing MAC/AE modes. We provide three case studies with concrete implementation results at Sections 6, 7, 8, and conclude at Section 9.

1.1 Related Work

Bitslice introduced by Biham [17] is a general technique to boost the software performance of serial symmetric-key algorithms by block-wise parallelization. It has been initially developed for DES, but it is also applicable to other primitives, such as tweakable block ciphers [33] and cryptographic permutations. Bitslicing AES has been extensively studied [8, 35, 44]. Matsuda and Moriai [34] showed how lightweight block ciphers can run fast with bitslicing on x86/x64 platforms. It also highlights the practical importance of fast, parallel implementation at the server side in the context of Internet-of-Things (IoT). The original form of bitslice assumes multiple input blocks, which can be few to n for n-bit block cipher.

The implementation strategy proposed by Bogdanov *et al.* [20] can be interpreted as bitslicing at the mode level. In principle it is applicable to any mode. For example, Bogdanov *et al.* reports 0.84 cycles per byte for CMAC on an Intel Haswell processor. However, it requires a certain number of multiple messages that processed at one time, which does not fit the typical interface of popular cryptographic libraries. Also the performance will be affected by the distribution of message length.

2 Preliminaries

Let $\{0,1\}^*$ be the set of all bit strings. For $x \in \{0,1\}^*$, $|x|$ denotes its bit length and $|x|_n$ for some positive integer n denotes $\lceil |x|/n \rceil$. The encryption function of a tweakable block cipher (TBC) is a function $E : \mathcal{K} \times \mathcal{T}_w \times \mathcal{M} \to \mathcal{M}$, where $\mathcal{K}$ is the key space, $\mathcal{T}_w$ is the tweak space, and $\mathcal{M}$ is the message space. The key is a secret value, and the tweak is typically used as a public value that may be determined by the users. We require that $E(K, T, *)$ for any $(K, T) \in \mathcal{K} \times \mathcal{T}_w$ is a permutation over $\mathcal{M}$. We may write $E(K, T, *)$ as $E_K^T(*)$ or $E_K(T, *)$. A block cipher is a TBC with $\mathcal{T}_w$ being a singleton, and in that case we drop $\mathcal{T}_w$ from the notation. The decryption function is denoted by $E^{-1} : \mathcal{K} \times \mathcal{T}_w \times \mathcal{M} \to \mathcal{M}$ such that $E_K^{-1}(T, E_K(T, M)) = M$ for any $(K, T, M) \in \mathcal{K} \times \mathcal{T}_w \times \mathcal{M}$. For $M \in \{0,1\}^*$, $(M[1], \ldots, M[m]) \xleftarrow{n} M$ for some positive integer n means a parsing of M into n-bit blocks, *i.e.*, $|M[i]| = n$ for $i = 1, \ldots, m-1$ and $|M[m]| \le n$, where $m = |M|_n$.

MAC and AE. We follow the standard syntax of MAC and AE schemes [13, 18, 30]. Let MAC be a deterministic MAC scheme. It consists of a tagging function

$\mathsf{MAC}.\mathcal{T} : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ and a verification function $\mathsf{MAC}.\mathcal{V} : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \{\top, \bot\}$, where $\mathcal{K}$ is the key space and $\mathcal{M}$ is the message space and $\mathcal{T}$ is the tag space. The tagging function is to compute the tag $T \in \mathcal{T}$ using key $K \in \mathcal{K}$ (which is shared by the prover and the verifier) and message $M \in \mathcal{M}$, and written as $T \leftarrow \mathsf{MAC}.\mathcal{T}_K(M)$. The prover sends the tuple (M, T) to the verifier. Once (M', T') is received (which may be tampered by the adversary), the verifier performs the verification function $\mathsf{MAC}.\mathcal{V}_K$, which first computes $\widehat{T} = \mathsf{MAC}.\mathcal{T}_K(M')$ and returns $\top$ (meaning verification success) if $T' = \widehat{T}$ and $\bot$ otherwise.

Let AE be a nonce-based AE scheme[7]. It consists of an encryption function $\mathsf{AE}.\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \to \mathcal{M} \times \mathcal{T}$ and a decryption function $\mathsf{AE}.\mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \times \mathcal{T} \to \mathcal{M} \cup \{\bot\}$. Here, $\mathcal{K}$ is the key space, $\mathcal{N}$ is the nonce space (which is the value that never repeats at encryption), $\mathcal{M}$ is the message space, $\mathcal{A}$ is the associated data (AD) space, and $\mathcal{T}$ is the tag space. AD is a part of input that is not encrypted but authenticated, for example the protocol header or the receiver address.

For encrypting $M \in \mathcal{M}$ with $A \in \mathcal{A}$ and $N \in \mathcal{N}$ and $K \in \mathcal{K}$, we compute $(C, T) \leftarrow \mathsf{AE}.\mathcal{E}_K(N, A, M)$ and send (N, A, C, T) to the receiver. After receiving the tuple (N', A', C', T') (which may be a tampered version of (N, A, C, T)), the receiver performs decryption by computing $\mathsf{AE}.\mathcal{D}_K(N', A', C', T')$ and if it returns $M' \neq \bot$ the receiver decides that the tuple is authenticated with the decrypted plaintext M'. Otherwise the receiver decides that the verification fails.

We omit the corresponding security notions defined with certain games, as they are less relevant to our work. Refer to [13, 18, 30] for them.

3 Pincer Verification of MAC Modes

Let $\mathsf{MAC} = (\mathsf{MAC}.\mathcal{T}, \mathsf{MAC}.\mathcal{V})$ be a deterministic MAC scheme. Let $M \in \mathcal{M}$ be a message and $\mathcal{S}$ be a internal state space of MAC. We require $\mathcal{S} = \mathcal{T}$. Suppose $\mathsf{MAC}.\mathcal{T}$ can be decomposed into two functions: $F : \mathcal{K} \times \mathcal{M} \times \mathcal{S} \to \mathcal{S}$ and $G : \mathcal{K} \times \mathcal{M} \times \mathcal{S} \to \mathcal{T}$ and an initial constant $\mathtt{init} \in \mathcal{S}$, such that

$$\mathsf{MAC}.\mathcal{T}_K(M) = G_K(M_2, F_K(M_1, \mathtt{init})), \tag{1}$$

where $M_1 \parallel M_2 = M$ determined by a certain parsing function $f : \mathcal{M} \to \mathcal{M} \times \mathcal{M}$. We write $(M_1, M_2) \xleftarrow{f} M$ to mean this parsing. The specification of f depends on how pincer verification is applied. Typically $|M_1|$ and $|M_2|$ are similar and $|M_1|$ is a multiple of n for the block length n. For simplicity we only consider such messages, however in practice this may not hold true (*e.g.*, a single-block message for $\mathsf{CBC\text{-}MAC}$). We can fall back to the normal verification procedure if the parsing is not possible. To implement pincer verification, we require that $G(K, M_2, *)$ is a TBC with tweak M_2 (and message space $\mathcal{S}$). The inverse of G is denoted by $G^{-1} : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \mathcal{S}$. We emphasize that we do not require a secure TBC for G: we just require $G(K, M_2, *)$ is a permutation for any (K, M_2).

[7] Our proposal covers AE schemes that do not necessarily require nonce, such as deterministic AE. We focus on nonce-based AE for simplicity.

Algorithm $\mathsf{MAC}.\mathcal{T}_K(M)$	**Algorithm** $\mathsf{MAC}.\mathcal{V}_K(M, T^*)$	**Algorithm** $\mathsf{MAC}.\mathcal{PV}_K(M, T^*)$
1. $(M_1, M_2) \leftarrow f(M)$	1. $(M_1, M_2) \leftarrow f(M)$	1. $(M_1, M_2) \leftarrow f(M)$
2. $S \leftarrow F(K, M_1, \texttt{Init})$	2. $S \leftarrow F(K, M_1, \texttt{Init})$	2. $S \leftarrow F(K, M_1, \texttt{Init})$
3. $T \leftarrow G(K, M_2, S)$	3. $T \leftarrow G(K, M_2, S)$	3. $S^* \leftarrow G^{-1}(K, M_2, T^*)$
4. **return** T	4. **return** $\top$ **if** $T = T^*$	4. **return** $\top$ **if** $S = S^*$
	5. **return** $\bot$ **otherwise**	5. **return** $\bot$ **otherwise**

Fig. 1: (Left) Tagging of a MAC function MAC. (Middle) Regular Verification. (Right) Pincer Verification.

When (M, T^*)[8] is received, the normal MAC verification is to check if $T \coloneqq \mathsf{MAC}.\mathcal{T}_K(M)$ is identical to T^*. Then, for $M = M_1 \parallel M_2$, we compute $S \leftarrow F(K, M_1, \texttt{Init})$ and $S^* \leftarrow G^{-1}(K, M_2, T^*)$ and check if $S = S^*$. Note that both F and G^{-1} can be computed in parallel, and the parsing function f determines which point to be matched in the entire MAC computation. To maximize the gain of parallelizability, this would be typically determined by the difference of computation costs of F and G^{-1}.

The soundness of pincer verification described above is simply shown by the following proposition.

Proposition 1. *If the above requirement holds for MAC, the normal verification (the middle of Fig. 1) and the pincer verification (the right of Fig. 1) are equivalent, that is, $\mathsf{MAC}.\mathcal{V}_K(M, T^*) = \mathsf{MAC}.\mathcal{PV}_K(M, T^*)$ for any (K, M, T^*).*

Proof. When (M, T^*) is given, let $S = F_K(M_1, \texttt{Init})$, $T = G_K(M_2, S)$ and $S^* = G_K^{-1}(M_2, T^*)$, where $M_1 \parallel M_2 \xleftarrow{f} M$. The normal verification accepts iff $T = T^*$ and the pincer verification accepts iff $S = S^*$. Since G is a TBC, the event $[T = T^*]$ is equivalent to $[G_K^{-1}(M_2, T) = G_K^{-1}(M_2, T^*)]$ which is identical to $[S = S^*]$. □

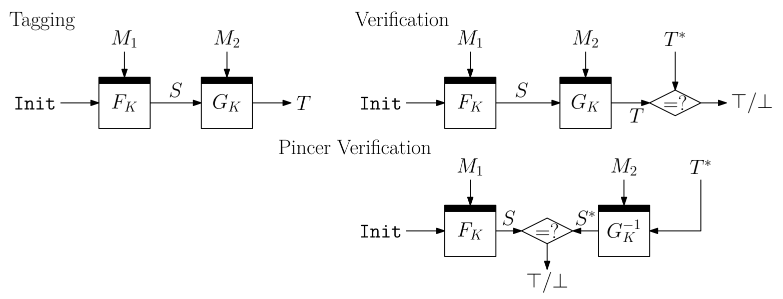

Fig. 2: MAC subroutines. The black rectangles in the boxes denote tweaks.

[8] We write (M, T^*) instead of (M', T') written at Section 2: this is convenient and intuitive to understand the data flow of pincer verification.

4 Pincer Verification of AE Modes

The idea of pincer verification is also extended to serial AE modes with some modifications. For this purpose we introduce an abstract model for serial AE. The idea is based on COFB [23]. For simplicity, we only consider the case that the matching point for the pincer decryption is in the ciphertext and not in the AD. First we define a component of (our abstraction of) serial AEs.

Definition 1. *Let $\widehat{E} : \mathcal{K} \times \mathcal{T}_w \times \mathcal{S} \to \mathcal{B} \times \mathcal{S}$ and $\mathcal{T}_w = \mathcal{W} \times \mathcal{B}$, for some finite sets $\mathcal{K}$, $\mathcal{W}$, $\mathcal{B}$ and $\mathcal{S}$. We say $\widehat{E}$ is a TBC with auxiliary output (TBC-AO) if there exist two functions, $\widehat{E}_f^{-1}, \widehat{E}_b^{-1} : \mathcal{K} \times \mathcal{T}_w \times \mathcal{S} \to \mathcal{B} \times \mathcal{S}$ such that, for any $(K, T_w = (W, B), B', S, S')$ fulfilling $\widehat{E}(K, T_w, S) = (B', S')$,*

$$\widehat{E}_f^{-1}(K, T'_w, S) = (B, S') \text{ and } \widehat{E}_b^{-1}(K, T'_w, S') = (B, S) \tag{2}$$

hold, where $T'_w = (W, B')$.

The functions $\widehat{E}$, $\widehat{E}_f^{-1}$ and $\widehat{E}_b^{-1}$ are required for encryption, (regular) decryption, and pincer decryption respectively. Eq. (2) implies that, (T'_w, S, S') is a tuple of (tweak, plaintext, ciphertext) of a TBC whose encryption and decryption routines are obtained by ignoring the first output elements of $\widehat{E}_f^{-1}$ and $\widehat{E}_b^{-1}$. As in the case of MAC, Definition 1 does not require any security property for TBC-AO.

Definition 2. *Let $\mathcal{T}_w = \mathcal{W} \times \mathcal{B}$, $\mathcal{W} = \mathcal{N} \times \mathcal{A}$, $\mathcal{B} = \mathcal{M}$ and $\mathcal{S} = \{0,1\}^n$ for nonce space $\mathcal{N}$, AD space $\mathcal{A}$, and message space $\mathcal{M}$. Let $\mathsf{SerialAE} = (\mathsf{SerialAE}.\mathcal{E}, \mathsf{SerialAE}.\mathcal{D})$ be an AE scheme that can be decomposed into two TBC-AOs[9] $F, G : \mathcal{K} \times \mathcal{T}_w \times \mathcal{S} \to \mathcal{M} \times \mathcal{S}$ as shown in Fig. 3 together with a parsing function f.*

Let us briefly explain Fig. 3. The encryption $\mathsf{SerialAE}.\mathcal{E}$ first parses M to M_1 and M_2 and encrypts M_1 to C_1 by F_K taking tweak $T_w = (N, A, M_1)$ and a constant `Init` as the initial state. The auxiliary output S is then used as the state input to G_K for encrypting M_2 to C_2 and producing the tag T. For (regular) decryption, $\mathsf{SerialAE}.\mathcal{D}$ first parses C to C_1 and C_2 and applies $F_{f,K}^{-1}$ to recover M_1 and the intermediate state S. This S is used as the state input to $G_{f,K}^{-1}$ to recover M_2 and the locally computed tag T to be checked.

For the sake of generality we consider both F and G take the nonce N and the entire AD A, however in the real schemes this is not necessary, say G may ignore A (which is the case of COFB).

The pincer decryption of SerialAE is derived in a similar manner to the case of MAC. When (N, A, C, T^*) is received, the pincer decryption routine $\mathsf{SerialAE}.\mathcal{PD}$ first parses C, and performs the forward computation in the same manner to the regular decryption using F_f^{-1} and the backward computation using G_b^{-1} in parallel. See Fig. 3. As well as the case of MAC, the parsing function f determines which point to be matched, so it should be determined by considering the efficiency gap between F_f^{-1} and G_b^{-1}.

Any SerialAE allows a pincer decryption, as follows.

[9] More precisely we only need F_f^{-1} for F.

```
Algorithm SerialAE.$\mathcal{E}(K, N, A, M)$

1. $(M_1, M_2) \leftarrow f(M)$
2. $(C_1, S) \leftarrow F_K((N, A, M_1), \mathtt{Init})$
3. $(C_2, T) \leftarrow G_K((N, A, M_2), S)$
4. $C \leftarrow (C_1 \,\|\, C_2)$
5. return $(C, T)$
```

```
Algorithm
SerialAE.$\mathcal{D}(K, N, A, C, T^*)$

1. $(C_1, C_2) \leftarrow f(C)$
2. $(M_1, S) \leftarrow F^{-1}_{f,K}((N, A, C_1), \mathtt{Init})$
3. $(M_2, T) \leftarrow G^{-1}_{f,K}((N, A, C_2), S)$
4. If $T = T^*$
5.     $M \leftarrow (M_1 \,\|\, M_2)$
6.     return $M$
7. Else return $\bot$
```

```
Algorithm
SerialAE.$\mathcal{PD}(K, N, A, C, T^*)$

1. $(C_1, C_2) \leftarrow f(C)$
2. $(M_1, S) \leftarrow F^{-1}_{f,K}((N, A, C_1), \mathtt{Init})$
3. $(M_2, S^*) \leftarrow G^{-1}_{b,K}((N, A, C_2), T^*)$
4. If $S = S^*$
5.     $M \leftarrow (M_1 \,\|\, M_2)$
6.     return $M$
7. Else return $\bot$
```

Fig. 3: Algorithms of Serial AE Scheme SerialAE.

Proposition 2. *The regular decryption and the pincer decryption (the middle and the right of Fig. 3) are equivalent for any* SerialAE*, that is,* SerialAE.$\mathcal{D}(K, N, A, C, T^*) =$ SerialAE.$\mathcal{PD}(K, N, A, C, T^*)$ *for any* (K, N, A, C, T^*).

Proof. The proof is basically the same as that of Proposition 1. We use the same notations as Fig. 3. Given (N, A, C, T^*) with key K, the derivations of S are identical for both SerialAE.$\mathcal{D}$ and SerialAE.$\mathcal{PD}$. We observe that $[T = T^*]$ is equivalent to $[S = S^*]$ as $G^{-1}_{f,K}$ and $G^{-1}_{f,K}$ is a pair of forward and backward permutation evaluations specified by (N, A, C_2) and K. Thus the verification result is always identical. If the verification successes, the decrypted plaintext $(M \leftarrow M_1 \,\|\, M_2)$ is identical; for M_1 it is trivial and for M_2, from Eq. (2), whenever $G^{-1}_{f,K}$ maps (C_2, S) to (M_2, T), with additional tweak variables (N, A), $G^{-1}_{b,K}$ maps (C_2, T) to (M_2, S). This concludes the proof.

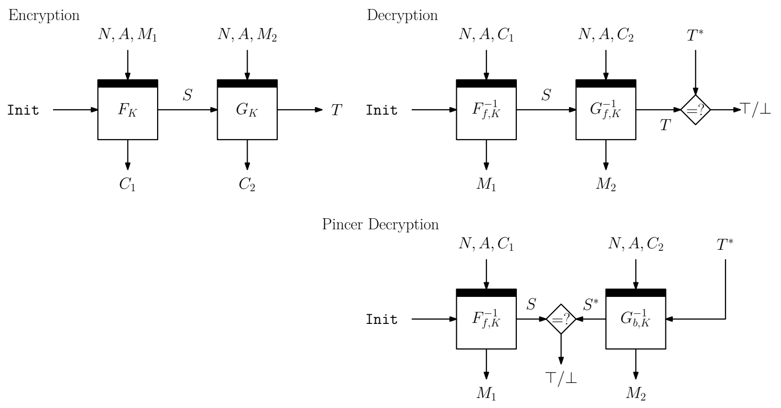

Fig. 4: AE subroutines. Black rectangles in the boxes denote tweaks, and the down arrows from the boxes denote auxiliary outputs.

5 Applications

5.1 MAC Modes

Pincer verification is applicable to any MAC scheme that can be interpreted as Eq. (1). This covers a wide range of MAC modes. Intuitively, one can apply when an internal chaining state of a MAC mode is computable from the tag. This requires that the tag is untruncated, which is usually a part of the specification; hereafter we assume untruncated tags when it is allowed in the specification. Most notably this applies to the classical CBC-MAC and CMAC, and their variants (See the last of Section 5). However, there are some exceptions because of their non-invertible state update (*e.g.*, GCBC [40]).

For parallel MAC modes, even if the pincer verification is applicable, the advantage is generally small because of their inherent parallelizability for regular verification. This applies to PMAC [42] or Carter-Wegman MACs (assuming a parallelizable universal hash function), such as GMAC [4] or Poly1305 [14]. Still some practical benefit is expected when the message is too short to gain the benefit of parallelizability. For example, PMAC for two message blocks essentially reduces to a variant of CBC-MAC and pincer verification will make it to two parallel block cipher calls instead of two serial calls. The main component of GMAC is the polynomial hash function over $\mathrm{GF}(2^n)$, which is defined as $\mathrm{poly}_K(M) = \sum_{i=1,\dots,m} M[i] \cdot K^{m-i+1}$ for m-block message $M = (M[1], \dots, M[m])$. This can be easily parallelized by pre-computing $K^2, \dots, K^{s+1}$ so that s blocks are processed in parallel [26]. The choice of s will depend on the platform and implementation of multiplication over $\mathrm{GF}(2^n)$. For $s = 2$, the pincer verification allows a 2-parallel evaluation of $\mathrm{poly}_K(M)$ without caching K^2. The benefit would be small though.

Sponges. A cryptographic permutation is also a popular primitive to build a MAC. The typical MAC construction is (keyed) Sponge, such as KMAC [5]. Unfortunately Sponge-based MAC usually truncates the tag to ensure security, as otherwise the adversary is able to derive the internal state that can be used to create a forgery. In case the permutation is used to instantiate an Even-Mansour (EM) cipher and that EM is used to built a MAC in a serial MAC mode, we generally do not need to truncate and pincer verification may be possible. Chaskey [37] is one such example.

5.2 AE Modes

Small-state nonce-based AE has been very actively studied in the context of lightweight cryptography. CLOC [31] is a CAESAR submission that explicitly claims their small state of $2n$ bits as one of their advantages (when n-bit block cipher is used, and we exclude the memory for the key). It needs two block cipher calls per one input block (rate 1/2). COFB [23, 24] is the first rate-1 block cipher mode for AE with a small state size of $1.5n$ bits. SAEB [38] further pushes the limit of state size to just n bits, which is the minimum. Its rate is effectively 1/2. To achieve these small state size, COFB and SAEB adopt serial structures. For

AE based on TBC, Romulus [29] and PFB [39] achieve the smallest state size, *i.e.*, the state size is what is needed to implement TBC itself. More precisely, the state size is $n+t$ bits, for a TBC of n-bit block and t-bit tweak (again we exclude the key).

As we have shown, pincer decryption is possible whenever the AE scheme is interpreted as SerialAE of Definition 1. We investigated state-of-the-art small-state modes mentioned above.

COFB. Fig. 5 shows an abstract, general structure of COFB and the related designs. Details are different by designs, but this figure is enough to discuss. The core component is the (keyless) state update function $\rho : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$. The n-bit block TBC $\widetilde{E}_K$ takes distinct tweaks for each call, and in case of COFB it is instantiated by a block cipher with a mask derived by a nonce, a key, and a block index. The ρ function is defined as $\rho(S, M) \to (S', C)$ (where the first argument corresponds to the state value) such that

$$S' = \mathsf{G}(S) \oplus M, \text{ and } C = M \oplus S \tag{3}$$

using an $n \times n$ binary matrix G. The function ρ is used to update the internal state (S) and to encrypt a plaintext block (M). When AD is processed, the first argument of ρ is an AD block and the second output is ignored (See Fig. 5). In the regular decryption we need to recover (S', M) from (S, C). This is possible irrespective of G by ρ_f^{-1}, which we call *forward ρ inversion*, defined as $\rho_f^{-1}(S, C) = (C \oplus S \oplus \mathsf{G}(S), C \oplus S)$. The correctness of the scheme is trivial to see, while we need some conditions on G to make it secure.

To make pincer verification possible for COFB, we need to recover (S, M) from (S', C). We call this procedure *backward ρ inversion.* It requires that $(\mathsf{G}+\mathsf{I})$ is regular, where I denotes the identity matrix. If this holds the pincer verification is implemented by using ρ_b^{-1} defined as

$$\rho_b^{-1}(S', C) = ((\mathsf{G}+\mathsf{I})^{-1}(C \oplus S'), (\mathsf{G}+\mathsf{I})^{-1}(C \oplus S') \oplus C). \tag{4}$$

The right hand side of the above equation coincides with (S, M). The specifications of COFB shown at [23] and [24] qualify the regularity of $(\mathsf{G}+\mathsf{I})$, and hence meet the condition of Definition 2. One can simply confirm this by backtracking the decryption algorithm from the tag, which involves (tweakable) block cipher decryption and ρ_b^{-1} in an alternating manner. However, the version that was submitted to NIST LWC, called GIFT-COFB [11], uses a simple G such that $\mathsf{G}+\mathsf{I}$ has rank $n-1$. We stress that this does not harm its provable security; the security claim requires that the ranks of G and $\mathsf{G}+\mathsf{I}$ are high but not necessarily n.

SAEB and SAEAES. SAEB [38] cannot use pincer decryption since the scheme is essentially identical to duplex sponge [16], which is a variant of CFB with an additional chaining state (rate part). The backtracking the internal state from the tag requires the knowledge of the plaintext rather than the ciphertext. A NIST LWC 2nd-round candidate SAEAES is an instantiation of SAEB, hence pincer decryption is not possible for it.

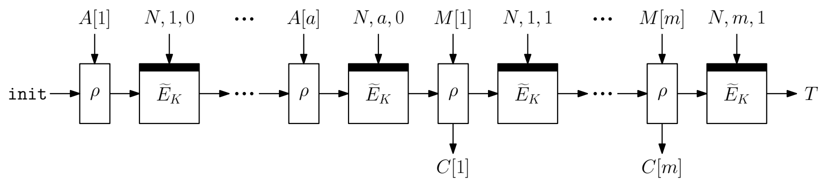

Fig. 5: General structure of COFB and the related designs, where ρ is the state update function. The second output of ρ is ignored while processing AD. $\widetilde{E}_K$ is a TBC possibly instantiated by a block cipher.

Romulus and PFB. Romulus [29] and PFB [39] share the general design shown at Fig. 5. They use TBCs. Their ρ functions have different structures from that of COFB (Eq. (3)). However, they allow the backward ρ inversion ρ_b^{-1}, hence pincer decryption is possible. The case of PFB is rather obvious, for its plaintext-feedback structure. The case of Romulus is bit more complex. Section 6 will give more details together with actual implementation results.

Summary of Applicability. We list MAC and AE modes that allow pincer verification/decryption as follows. We stress that the list is not exhaustive as it is not possible to list all the known modes in the literature. For CCM, we refer to Section 8.

- MAC modes: CMAC [3,30], CBC-MAC, EMAC [22], ECBC, FCBC, XCBC [18], TMAC [32], CBCR [46], PC-MAC [36], Chaskey [37]
- AE modes: CCM [2], Versions of COFB [23,24], Romulus [29], PFB [39]

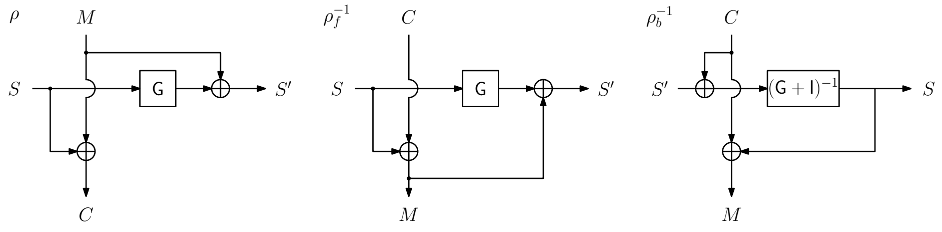

Fig. 6: The state update functions of COFB.

6 Case Study 1: Romulus

6.1 Pincer Verification of Romulus

We show a pincer decryption implementation for Romulus on microcontrollers. First we need to show that pincer decryption is indeed possible. In this paper, we focus on the nonce-based variant of $\mathsf{Romulus}$, $\mathsf{Romulus\text{-}N}$, which is the main variant of the NIST LWC submission. The pseudocode is shown in Fig. 11 at Appendix A. It uses SKINNY TBC [12]. We skip the details and focus on its ρ. For simplicity we only show the case that the last message block is n bits, but this can be extended to the case that the last block is less than n bits. As mentioned, the definition of ρ is different from Eq. (3). We have

$$\rho(S, M) = (S', C), \text{ such that } S' = S \oplus M, C = M \oplus \mathsf{G}(S)$$

for some $n \times n$ binary matrix G (see Fig. 7). The regular decryption is done by ρ_f^{-1}, which corresponds to ρ^{-1} defined as $\rho_f^{-1}(S, C) = (\mathsf{G}(S) \oplus C \oplus S, \mathsf{G}(S) \oplus C)$. To make pincer decryption possible, we need ρ_b^{-1} that recovers (S, M) given (S', C), and it can be defined as

$$\rho_b^{-1}(S', C) \coloneqq ((\mathsf{G} + \mathsf{I})^{-1}(S' \oplus C), (\mathsf{G} + \mathsf{I})^{-1}(S' \oplus C)) \oplus S'$$

if $\mathsf{G} + \mathsf{I}$ is regular. The ρ of $\mathsf{Romulus}$ fulfills this condition.

6.2 ESP32

We chose ESP32 for our implementation target, which is a very popular low-cost 32-bit microcontroller with buit-in Wi-Fi and Bluetooth. It is widely used for IoT applications. It shipments reached 100-Million at 2017 [7] and a report in 2018 showed that ESP became a leader in the embedded WiFi chip market sector [6]. ESP32 uses Tensilica's Xtensa LX6 microprocessor and is available in dual-core and single-core versions. We took Sipeed's Maixduino[10] which contains a dual-core ESP32 running at 240 MHz. This board was also used by a comprehensive microcontroller benchmark of NIST LWC candidates including Romulus, conducted by Renner, Pozzobon, and Mottok [41] (see also `https://lwc.las3.de/`).

The ESP32 chip used in our implementation has two cores, CPU0 and CPU1. Normally, the user application task runs on CPU1, while the system task runs on CPU0. The ESP32 runs FreeRTOS built in, and this allows tasks to run on different cores with a small overhead. CPU1 and CPU0 share the same memory space. Therefore, by accessing the same memory address in the task on CPU1 and the task on CPU0, coordinated processing can be performed between tasks running on multiple cores.

6.3 Implementation Details

In our implementation, the decryption routine first parses C into C_1 and C_2 and runs CPU1 to compute the forward direction of pincer decryption taking C_1, and CPU0 to compute the backward direction taking C_2 and the received tag T^*. For simplicity, we let C_1 and C_2 have (almost) equal number of blocks, thus the

[10] `https://maixduino.sipeed.com/en/`

check is done at the midpoint of message, and assume the empty associated data. The information necessary for a task on CPU0 to operate is achieved by passing the memory location where the information is stored when the task starts. This keeps the API of decryption as normal (single-core) decryption, hence users are not required to change the outer code.

Due to the overhead of task switch (*i.e.*, invoking another task at CPU1 while CPU0 runs a main task), the pincer decryption is disadvantageous when the message is too short. In our environment, a task switch requires around 9,000 cycles. As we will see later, this makes pincer decryption effective for messages longer than 4 blocks (64 bytes). We wrote a C code and complied it with `xtensa-esp32-elf-gcc 5.2.0`.

Implementation Results. Table 1 shows our implementation results on ESP32. When messages are long enough (1,536 bytes in our case), pincer decryption roughly halves the cycles from the regular decryption, showing pincer decryption performs ideally at this setting. The performance of encryption is mostly the same as the regular decryption. Since our purpose is to show the feasibility, we adopt a fast, table-based implementation for SKINNY. If cache-timing attacks are concern one can adopt a constant-time implementation, such as [8]. Other implementation details of SKINNY are shown in Appendix B.

Table 1: Comparison of regular decryption and pincer decryption of Romulus, measured by cycles per byte.

Message length (byte)	16	64	128	256	512	1024	1536
Regular Decryption	1418	705	586	526	500	485	480
Pincer Decryption	1427	615	425	330	283	261	254

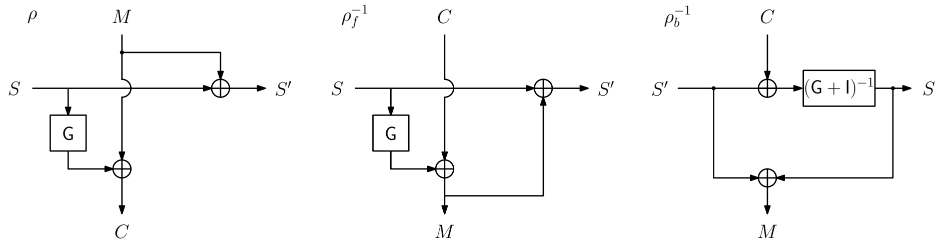

Fig. 7: The state update functions of Romulus.

Table 2: Comparison of regular verification and pincer verification of CMAC, measured by cycle per byte. Measurements were taken on single core.

Message length (byte)	64	128	192	256	512	1024
Regular verification	2.00	2.34	2.68	2.85	3.10	3.35
Pincer verification	1.84	1.52	1.66	1.63	1.65	1.65

7 Case Study 2: CMAC

The second case study takes CMAC on x86 CPUs with AES-NI. We implemented the regular and the pincer verifications of CMAC showed in Fig. 8. The regular one was implemented in the (trivial) serial manner, and the pincer one was implemented by parallelizing the lines 3–5 and the lines 6–12. In detail, we perform one-block encryption and one-block decryption of AES-128 by interleaving `aesenc` and `aesdec` instructions. This effectively parallelizes two-block computations, so we can process one-block encryption and decryption with the cycles to process one-block encryption/decryption. The technique is common for parallelizable modes, however previous implementations interleave multiple `aesenc` *or* `aesdec` instructions. The benchmark environment is as follows:

- OS: Linux version 4.15.0-128-generic
- Distribution: Ubuntu 18.04.5 LTS
- Processor: Intel Xeon Silver 4114 (2.2GHz Skylake), Turboboost off
- Language: C with intrinsics
- Compiler: gcc 7.5.0

Table 2 shows the results of our implementations. Note that both results were taken on single core. In our environment, each AES instruction has latency 4 and throughput 1, in cycles. It shows that the cycle per byte (cpb) of the pincer verification converges to the half of that of the regular verification when the message is long enough. For a long message, the regular verification performance is almost identical to the that of single AES call. The use of intrinsics and some C function calls in our code introduced some overhead for both implementations, which could be improved by the use of assembly language. In any case, the important point is that we halved the verification time on single core.

8 Case Study 3: CCM

We show how pincer verification technique is applied to CCM, which is one of the two NIST recommended AE modes based on AES (Figure 10). It is widely deployed by major protocols and products (TLS, IPSec, WPA2, and lots of cryptographic library). Compared with another NIST recommendation GCM, its efficiency is generally inferior on modern desktop/mobile platforms. However, CCM does not need a 128-bit multiplier that is used by GCM, hence it enables a simpler implementation with a small code. This makes CCM useful on constrained devices.

```
Algorithm CMAC.$\mathcal{V}_K(M, T^*)$

 1. $T \leftarrow 0^n$, $L_1 \leftarrow 2E_K(0^n)$, $L_2 \leftarrow 2L_1$
 2. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
 3. for $i = 1$ to $m - 1$
 4.     $T \leftarrow E_K(T \oplus M[i])$
 5. end for
 6. if $|M[m]| = n$ then
 7.     $T \leftarrow E_K(T \oplus L_1 \oplus M[m])$
 8. else $T \leftarrow E_K(T \oplus L_2 \oplus \mathtt{pad}(M[m]))$
 9. if $T = T^*$ then return $\top$
10. else return $\perp$
```

```
Algorithm CMAC.$\mathcal{PV}_K(M, T^*)$

 1. $S \leftarrow 0^n$, $L_1 \leftarrow 2E_K(0^n)$, $L_2 \leftarrow 2L_1$
 2. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
 3. for $i = 1$ to $m/2$
 4.     $S \leftarrow E_K(S \oplus M[i])$
 5. end for
 6. $S^* \leftarrow E_K^{-1}(T^*)$
 7. if $|M[m]| = n$ then
 8.     $S^* \leftarrow S^* \oplus L_1 \oplus M[m]$
 9. else $S^* \leftarrow S^* \oplus L_2 \oplus \mathtt{pad}(M[m]))$
10. for $i = m - 1$ to $m/2 + 1$
11.     $S^* \leftarrow E_K^{-1}(S^*) \oplus M[i]$
12. end for
13. if $S = S^*$ then return $\top$
14. else return $\perp$
```

Fig. 8: (Left) Regular verification of CMAC. (Right) Pincer verification of CMAC. The padding is $\mathtt{pad}(x) = x \,\|\, 10^{n-|x|-1}$ when $0 \leq |x| < n$ and $\mathtt{pad}(x) = x$ when $|x| = n$. $2L$ denotes GF(2^n) multiplication by the constant $\mathtt{x}$. The pincer verification assumes even m for simplicity.

The overall structure of CCM is MAC-then-Encrypt, where MAC is a plain CBC-MAC and encryption is CTR mode. While CTR mode is parallelizable it can be seen as SerialAE[11], and pincer verification is possible. Suppose we decrypt a tuple (N, A, C, T^*) with CCM, where $|A|_n = a$ and $|C|_n = m$ and $n = 128$, the normal procedure is to first decrypt C by CTR mode decryption to obtain the (unverified) plaintext M, and compute the candidate tag T by CBC-MAC taking (N, A, M) and check if $T = T^*$. CTR decryption and CBC-MAC are performed over m and $a + m$ blocks, hence we need $a + 2m$ block cipher calls in total. We also need 2 or 3 more calls depending on the parameters.

Pincer Decryption of CCM. Suppose we can process s blocks of AES computations in parallel, how fast CCM can decrypt? The trivial solution (which we call plain parallel) is to first perform CTR decryption in s-parallel, and perform CBC-MAC in serial. It costs $\lceil m/s \rceil + a + m$ parallel AES calls for a AD blocks and m ciphertext blocks. A more advanced solution is to interleave CTR decryption and CBC-MAC, which we call (plain) interleave. It reduces the latency of CTR decryption and has been employed by OpenSSL[12]. Our proposal is to combine interleaving and pincer verification of CBC-MAC. Depending on s, we derive two strategies. The first one (Option 1) interleaves CTR decryption and (the plain, forward direction of) CBC-MAC, and once CTR decryption is done, continues the pincer verification for the remaining CBC-MAC inputs. The second one (Option 2)

[11] Strictly speaking it is not, because of the specification of `encode`, however this difference is not critical.

[12] `https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aesni-x86_64.pl`

Table 3: Comparison of CCM decryption costs for a AD blocks, m ciphertext blocks, using parallel computation of s AES blocks. We ignore constant overhead.

Plain parallel	Interleave	Pincer decrypt (Opt. 1)	Pincer decrypt (Opt. 2)
$\lceil m/s \rceil + a + m$	$a + m$	$\lceil (a + m + \lceil m/(s-1) \rceil)/2 \rceil$	$\lceil (a+m)/2 \rceil$

performs "pincer decryption" of CTR, that is, from the first ciphertext block (forward direction) and from the last ciphertext block (backward direction), and at the same time performs the pincer verification of CBC-MAC. The latter is possible as above CTR decryption gives necessary input blocks. When CTR decryption is done, it continues the pincer verification for the remaining CBC-MAC blocks. See Figure 9 for an illustration of Option 2. Option 1 works when $s \geq 2$, while Option 2 is faster but needs $s \geq 4$: two for pincer CTR decryption and two for pincer verification of CBC-MAC. Clearly $s = 4$ is optimal and achieves about $\lceil (a+m)/2 \rceil$ calls. When $s = 4$, (plain parallel, interleave, pincer verification option 1, pincer verification option 2) needs about $(a + 5m/4, a + m, a/2 + 2m/3, (a + m)/2)$ calls, and when $s = 8$, $(a + 9m/8, a + m, a/2 + 4m/7, (a + m)/2)$ calls.

Table 4 shows the implementation results of CCM with AES-NI, using the same environment as Section 7. We set $s = 4$ as the best choice from the latency figures of AES-NI on our platform. We set 64-bit nonce and empty AD. The results showed the same trend as CMAC (Section 7): for a long message the pincer decryption (Option 2) halves the computation cost from the previously known strategy (interleave).

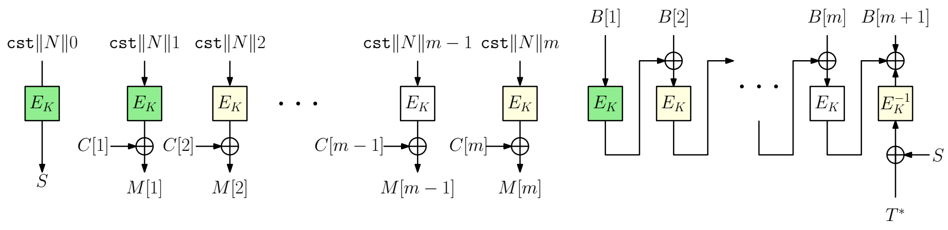

Fig. 9: Pincer decryption (Option 2) for CCM with $s = 4$ and empty AD. $B[1]$ involves N, and $B[i] = M[i-1]$ for $i > 1$. After the green boxes are computed, 4 light yellow boxes consisting of 3 E_K and 1 E_K^{-1} are computed in parallel.

9 Conclusion

This article presented a new parallel implementation strategy, pincer verification, for verification/decryption routines of serial MAC and AE modes. The core idea is pretty simple, yet to our knowledge it has not been proposed in the

```
Algorithm $\mathsf{CCM}.\mathcal{E}_K(N, A, M)$

1. $B \leftarrow \mathsf{encode}(N, A, M)$
2. $U \leftarrow \mathsf{CBC\text{-}MAC}[E_K](B)$
3. $C \leftarrow \mathsf{CTR}[E_K](N, M)$
4. $T \leftarrow E_K(\mathsf{cst} \,\|\, N \,\|\, \langle 0 \rangle_{8\lambda}) \oplus U$
5. return $(C, T)$
```

```
Algorithm $\mathsf{CCM}.\mathcal{D}_K(N, A, C, T^*)$

1. $M \leftarrow \mathsf{CTR}[E_K](N, C)$
2. $B \leftarrow \mathsf{encode}(N, A, M)$
3. $U \leftarrow \mathsf{CBC\text{-}MAC}[E_K](B)$
4. $T \leftarrow E_K(\mathsf{cst} \,\|\, N \,\|\, \langle 0 \rangle_{8\lambda}) \oplus U$
5. if $T = T^*$ then return $M$
6. else return $\perp$
```

```
Algorithm $\mathsf{CBC\text{-}MAC}[E_K](X)$

1. $S \leftarrow 0^n$
2. $(X[1], \ldots, X[x]) \overset{n}{\leftarrow} X$
3. for $i = 1$ to $x$
4.     $S \leftarrow E_K(S \oplus X[i])$
5. return $S$
```

```
Algorithm $\mathsf{CTR}[E_K](N, X)$

1. $(X[1], \ldots, X[x]) \overset{n}{\leftarrow} X$
2. for $i = 1$ to $x$
3.     $Y[i] \leftarrow E_K(\mathsf{cst} \,\|\, N \,\|\, \langle i \rangle_{8\lambda}) \oplus X[i]$
4. $Y \leftarrow Y[1] \,\|\, \ldots \,\|\, Y[x]$
5. return $Y$
```

Fig. 10: Algorithms of CCM. encode is an encoding function, cst is a byte determined by λ, and λ determines the maximum message length. $\langle i \rangle_j$ denotes the j-bit encoding of integer $i > 0$. Nonce is $15 - \lambda$ bytes, where $n = 128$. See [2] for details.

Table 4: Comparison of regular decryption (which implements interleaving of CTR decryption and CBC-MAC) and pincer decryption (Option 2 with $s = 4$) for CCM, measured by cycle per byte. Measurements were taken on single core.

Message length (byte)	64	128	192	256	512	1024
Regular decryption	3.97	3.28	3.29	3.30	3.30	3.51
Pincer decryption	3.59	2.53	2.24	2.11	1.90	1.78

literature. Although it is intuitive for the case of simple mode such as CBC-MAC, to deal with more complex mode we provided a formalization to see when and how pincer verification is possible that maintains the original functionality and security. Our case studies demonstrated its practicality. Notably it improves the verification/decryption performance of two NIST recommendations (CMAC and CCM) for x86 CPUs by a factor of 2.

We list some final remarks for future research. For CMAC and CCM, experiments with 64-bit ARM CPUs with AES instructions will be interesting from a practical point of view. The applicability of pincer decryption can be extended to some more generic-composition-like AE schemes, such as SIV [43], but whether it is effective or not depends on the MAC and Encryption parts. Although dual/multi-core microcontrollers are becoming popular, the implementation is still bit tricky and needs ad-hoc analysis to get the best performance. Finally, it would be interesting to consider if bitslice is combined with pincer verification, to improve performance on single core. Bitslice is usually for encryption of parallel blocks, however if encryption and decryption routines have a great similarity (*e.g.*, Feistel), it may be possible to bitslice one encryption and one decryption of a cipher in an efficient manner.

Acknowledgements

The authors would like to thank the anonymous reviewers of SAC 2021 for their insightful comments, and thank Thomas Peyrin and Mustafa Khairallah for feedback.

References

[1] NIST Lightweight Cryptography, `https://csrc.nist.gov/projects/lightweight-cryptography`, National Institute of Standards and Technology
[2] Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST Special Publication 800-38C (2004), national Institute of Standards and Technology
[3] Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication. NIST Special Publication 800-38B (2005), national Institute of Standards and Technology
[4] Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D (2007), national Institute of Standards and Technology
[5] SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. NIST Special Publication 800-185 (2016), national Institute of Standards and Technology
[6] 2017 wireless connectivity market analysis. TSR Report (2018), `www.t-s-r.co.jp/e/report/4543.html`
[7] Espressif milestones (2019), `www.espressif.com/en/company/about-us/milestones`
[8] Adomnicai, A., Peyrin, T.: Fixslicing AES-like ciphers. IACR TCHES **2021**(1), 402–425 (2021). https://doi.org/10.46586/tches.v2021.i1.402-425, `https://tches.iacr.org/index.php/TCHES/article/view/8739`
[9] Avanzi, R.: The QARMA block cipher family. IACR Trans. Symm. Cryptol. **2017**(1), 4–44 (2017). https://doi.org/10.13154/tosc.v2017.i1.4-44
[10] Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part II. LNCS, vol. 9453, pp. 411–436. Springer, Heidelberg (Nov / Dec 2015). https://doi.org/10.1007/978-3-662-48800-3_17
[11] Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT-COFB. Submission to NIST Lightweight Cryptography (2019)
[12] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part II. LNCS, vol. 9815, pp. 123–153. Springer, Heidelberg (Aug 2016). https://doi.org/10.1007/978-3-662-53008-5_5
[13] Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 531–545. Springer, Heidelberg (Dec 2000). https://doi.org/10.1007/3-540-44448-3_41
[14] Bernstein, D.J.: The poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 32–49. Springer, Heidelberg (Feb 2005). https://doi.org/10.1007/11502760_3

[15] Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.X., Todo, Y., Viguier, B.: Gimli : A cross-platform permutation. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 299–320. Springer, Heidelberg (Sep 2017). https://doi.org/10.1007/978-3-319-66787-4_15

[16] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: Single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-28496-0_19

[17] Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE'97. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (Jan 1997). https://doi.org/10.1007/BFb0052352

[18] Black, J., Rogaway, P.: CBC MACs for arbitrary-length messages: The three-key constructions. Journal of Cryptology **18**(2), 111–131 (Apr 2005). https://doi.org/10.1007/s00145-004-0016-3

[19] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (Sep 2007). https://doi.org/10.1007/978-3-540-74735-2_31

[20] Bogdanov, A., Lauridsen, M.M., Tischhauser, E.: Comb to pipeline: Fast software encryption revisited. In: Leander, G. (ed.) FSE 2015. LNCS, vol. 9054, pp. 150–171. Springer, Heidelberg (Mar 2015). https://doi.org/10.1007/978-3-662-48116-5_8

[21] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knežević, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (Dec 2012). https://doi.org/10.1007/978-3-642-34961-4_14

[22] Bosselaers, A., Preneel, B. (eds.): Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040, Lecture Notes in Computer Science, vol. 1007. Springer (1995)

[23] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based authenticated encryption: How small can we go? In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 277–298. Springer, Heidelberg (Sep 2017). https://doi.org/10.1007/978-3-319-66787-4_14

[24] Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based authenticated encryption: How small can we go? Journal of Cryptology **33**(3), 703–741 (Jul 2020). https://doi.org/10.1007/s00145-019-09325-z

[25] Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon. Submission to NIST Lightweight Cryptography (2019)

[26] Gueron, S., Kounavis, M.E.: Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Inf. Process. Lett. **110**(14-15), 549–553 (2010)

[27] Guo, C., khairallah, M., Minematsu, K., Peyrin, T.: Romulus v1.3. Submission to NIST Lightweight Cryptography (2021)

[28] Hoffert, J.D.S., Peeters, M., Assche, G.V., Keer, R.V., Mella, S.: Zoodyak. Submission to NIST Lightweight Cryptography (2019)

[29] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Duel of the titans: The Romulus and Remus families of lightweight AEAD algorithms. IACR Trans. Symm. Cryptol. **2020**(1), 43–120 (2020). https://doi.org/10.13154/tosc.v2020.i1.43-120

[30] Iwata, T., Kurosawa, K.: OMAC: One-key CBC MAC. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 129–153. Springer, Heidelberg (Feb 2003). https://doi.org/10.1007/978-3-540-39887-5_11

[31] Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Authenticated encryption for short input. In: Cid, C., Rechberger, C. (eds.) FSE 2014. LNCS, vol. 8540, pp. 149–167. Springer, Heidelberg (Mar 2015). https://doi.org/10.1007/978-3-662-46706-0_8

[32] Kurosawa, K., Iwata, T.: TMAC: Two-key CBC MAC. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 33–49. Springer, Heidelberg (Apr 2003). https://doi.org/10.1007/3-540-36563-X_3

[33] Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 31–46. Springer, Heidelberg (Aug 2002). https://doi.org/10.1007/3-540-45708-9_3

[34] Matsuda, S., Moriai, S.: Lightweight cryptography for the cloud: Exploit the power of bitslice implementation. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 408–425. Springer, Heidelberg (Sep 2012). https://doi.org/10.1007/978-3-642-33027-8_24

[35] Matsui, M.: How far can we go on the x64 processors? In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 341–358. Springer, Heidelberg (Mar 2006). https://doi.org/10.1007/11799313_22

[36] Minematsu, K., Tsunoo, Y.: Provably secure MACs from differentially-uniform permutations and AES-based implementations. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 226–241. Springer, Heidelberg (Mar 2006). https://doi.org/10.1007/11799313_15

[37] Mouha, N., Mennink, B., Van Herrewege, A., Watanabe, D., Preneel, B., Verbauwhede, I.: Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. In: Joux, A., Youssef, A.M. (eds.) SAC 2014. LNCS, vol. 8781, pp. 306–323. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-319-13051-4_19

[38] Naito, Y., Matsui, M., Sugawara, T., Suzuki, D.: SAEB: A lightweight blockcipher-based AEAD mode of operation. IACR TCHES **2018**(2), 192–217 (2018). https://doi.org/10.13154/tches.v2018.i2.192-217, `https://tches.iacr.org/index.php/TCHES/article/view/885`

[39] Naito, Y., Sugawara, T.: Lightweight authenticated encryption mode of operation for tweakable block ciphers. IACR TCHES **2020**(1), 66–94 (2019). https://doi.org/10.13154/tches.v2020.i1.66-94, `https://tches.iacr.org/index.php/TCHES/article/view/8393`

[40] Nandi, M.: Fast and secure CBC-type MAC algorithms. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 375–393. Springer, Heidelberg (Feb 2009). https://doi.org/10.1007/978-3-642-03317-9_23

[41] Renner, S., Pozzobon, E., Mottok, J.: A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In: Meng, W., Gollmann, D., Jensen, C.D., Zhou, J. (eds.) ICICS 20. LNCS, vol. 11999, pp. 495–509. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-61078-4_28

[42] Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 16–31. Springer, Heidelberg (Dec 2004). https://doi.org/10.1007/978-3-540-30539-2_2

[43] Rogaway, P., Shrimpton, T.: A provable-security treatment of the key-wrap problem. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 373–390. Springer, Heidelberg (May / Jun 2006). https://doi.org/10.1007/11761679_23

[44] Schwabe, P., Stoffelen, K.: All the AES you need on Cortex-M3 and M4. In: Avanzi, R., Heys, H.M. (eds.) SAC 2016. LNCS, vol. 10532, pp. 180–194. Springer, Heidelberg (Aug 2016). https://doi.org/10.1007/978-3-319-69453-5_10

[45] Wang, B., Gu, X., Yang, Y.: Saber on ESP32. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 20, Part I. LNCS, vol. 12146, pp. 421–440. Springer, Heidelberg (Oct 2020). https://doi.org/10.1007/978-3-030-57808-4_21

[46] Zhang, L., Wu, W., Zhang, L., Wang, P.: CBCR: CBC MAC with rotating transformations. SCIENCE CHINA Information Sciences **54**(11), 2247–2255 (2011)

A The pseudocode of Romulus

We present the pseudocode of Romulus (more precisely the nonce-based variant of Romulus, Romulus-N) in Fig. 11.

```
**Algorithm** Romulus-N.Enc$_K(N, A, M)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** $n$
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** $24$
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7.     $(S, \eta) \leftarrow \rho(S, A[2i-1])$
8.     $S \leftarrow \widetilde{E}_K^{(A[2i], 8, \overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N, w_A, \overline{a})}(S)$
13. $(M[1], \ldots, M[m]) \xleftarrow{n} M$
14. **if** $|M[m]| < n$ **then** $w_M \leftarrow 21$ **else** $20$
15. **for** $i = 1$ **to** $m - 1$
16.     $(S, C[i]) \leftarrow \rho(S, M[i])$
17.     $S \leftarrow \widetilde{E}_K^{(N, 4, \overline{i})}(S)$
18. **end for**
19. $M'[m] \leftarrow \mathtt{pad}_n(M[m])$
20. $(S, C'[m]) \leftarrow \rho(S, M'[m])$
21. $C[m] \leftarrow \mathtt{lmt}_{|M[m]|}(C'[m])$
22. $S \leftarrow \widetilde{E}_K^{(N, w_M, \overline{m})}(S)$
23. $(\eta, T) \leftarrow \rho(S, 0^n)$
24. $C \leftarrow C[1] \,\|\, \ldots \,\|\, C[m-1] \,\|\, C[m]$
25. **return** $(C, T)$
```

```
**Algorithm** Romulus-N.Dec$_K(N, A, C, T)$

1. $S \leftarrow 0^n$
2. $(A[1], \ldots, A[a]) \xleftarrow{n,t} A$
3. **if** $a \bmod 2 = 0$ **then** $u \leftarrow t$ **else** $n$
4. **if** $|A[a]| < u$ **then** $w_A \leftarrow 26$ **else** $24$
5. $A[a] \leftarrow \mathtt{pad}_u(A[a])$
6. **for** $i = 1$ **to** $\lfloor a/2 \rfloor$
7.     $(S, \eta) \leftarrow \rho(S, A[2i-1])$
8.     $S \leftarrow \widetilde{E}_K^{(A[2i], 8, \overline{2i-1})}(S)$
9. **end for**
10. **if** $a \bmod 2 = 0$ **then** $V \leftarrow 0^n$ **else** $A[a]$
11. $(S, \eta) \leftarrow \rho(S, V)$
12. $S \leftarrow \widetilde{E}_K^{(N, w_A, \overline{a})}(S)$
13. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
14. **if** $|C[m]| < n$ **then** $w_C \leftarrow 21$ **else** $20$
15. **for** $i = 1$ **to** $m - 1$
16.     $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$
17.     $S \leftarrow \widetilde{E}_K^{(N, 4, \overline{i})}(S)$
18. **end for**
19. $\widetilde{S} \leftarrow (0^{|C[m]|} \,\|\, \mathtt{rmt}_{n-|C[m]|}(G(S)))$
20. $C'[m] \leftarrow \mathtt{pad}_n(C[m]) \oplus \widetilde{S}$
21. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$
22. $M[m] \leftarrow \mathtt{lmt}_{|C[m]|}(M'[m])$
23. $S \leftarrow \widetilde{E}_K^{(N, w_C, \overline{m})}(S)$
24. $(\eta, T^*) \leftarrow \rho(S, 0^n)$
25. $M \leftarrow M[1] \,\|\, \ldots \,\|\, M[m-1] \,\|\, M[m]$
26. **if** $T^* = T$ **then return** $M$ **else** $\perp$
```

```
**Algorithm** $\rho(S, M)$

1. $C \leftarrow M \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** $(S', C)$
```

```
**Algorithm** $\rho^{-1}(S, C)$

1. $M \leftarrow C \oplus G(S)$
2. $S' \leftarrow S \oplus M$
3. **return** $(S', M)$
```

Fig. 11: Romulus-N [29]. $\mathtt{lmt}_i(x)$ ($\mathtt{rmt}_i(x)$) denotes the leftmost (rightmost) i bits of x. See [29] for the definitions of padding, parsing and constants.

B Implementation Details of SKINNY

Romulus uses SKINNY (of its 128-bit block, 384-bit tweakey version) as the internal TBC. It has been reduced to 40 rounds from the original 56 [12], considering the large margin in security and the performance improvement in the final round specification [27]. Our implementation adopted this 40-round version, however it is straightforward to deduce the performance figures of 56-round case.

We took some efforts to improve the performance of SKINNY under Romulus; it would be adequate here to describe some of them in detail. First, we precompute the results of SKINNY's tweakey schedule and store them at RAM, which needs 704 bytes. This particularly saves on-line computation regarding TK3, a part of tweakey which just contains the secret key. TK2 computation is also saved, as it contains nonce and is identical to any encryption of plaintext block. TK1 computation is needed for every round. However, from the fact that it has cycle 16 (*i.e.*, the original TK1 value is recovered after 16 rounds) and that the last 8 bytes of TK1 are fixed to 0, TK1 computation is needed only for $(1, 3, 5, 7, 9, 11, 13, 15)$ rounds. The results can be reused for other rounds. Inside Romulus, SKINNY has 8-bit S-boxes (SubCells). We implement these S-boxes by look up tables (LUTs) because it is faster than the state-of-the-art constant-time implementation by Adomnicai and Peyrin [8], and the decryption code is easier to write. We pack 4 S-box output bytes into a 32-bit word, and we do not need to explicitly implement Shiftrows as it is absorbed in packing of bytes (which consists of three shifts and three XORs), for example:

```
#define SBOX_8(w)                \
  t0 = (w) & 0xff;               \
  t1 = (w >> 8) & 0xff;          \
  t2 = (w >> 16) & 0xff;         \
  t3 = (w >> 24);                \
  t0 = sbox[t0];                 \
  t1 = sbox[t1];                 \
  t2 = sbox[t2];                 \
  t3 = sbox[t3];                 \
  w = (t0 << 8)  ^               \
      (t1 << 16) ^               \
      (t2 << 24) ^               \
      (t3);
```

Tweakey schedule involves a byte permutation defined as

$$P_T = [9,\ 15,\ 8,\ 13,\ 10,\ 14,\ 12,\ 11,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7].$$

This can be decomposed into a sequence of 8-byte permutation, in different patterns for even and odd rounds. We implement $\rho_b^{-1}(S', C)$ following Eq. (4). We need to implement the inverse of MixColumn matrix M, which is

$$\mathsf{M}^{-1} = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,1\,1\,1 \\ 0\,1\,0\,1 \\ 1\,0\,0\,1 \end{pmatrix}.$$

A word-wise implementation of M^{-1} is written as

$$\begin{pmatrix} 0\,1\,0\,0 \\ 0\,1\,1\,1 \\ 0\,1\,0\,1 \\ 1\,0\,0\,1 \end{pmatrix} \times \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_1 \oplus w_2 \oplus w_3 \\ w_1 \oplus w_3 \\ w_0 \oplus w_3 \end{pmatrix}.$$

This shows that we need 5 instructions to implement M^{-1}, which is identical to the case of M. The remaining components are implemented in the same as in the encryption. Consequently, the decryption of SKINNY is implemented in roughly the same number of instructions as encryption.