ELM : A Low-Latency and Scalable Memory Encryption Scheme

Akiko Inoue[1], Kazuhiko Minematsu[1], Maya Oda[2], Rei Ueno[2], and Naofumi Homma[2]

[1] NEC, Kawasaki, Japan
a_inoue@nec.com, k-minematsu@nec.com
[2] Research Institute of Electrical Communication, Tohoku University,
odam@riec.tohoku.ac.jp, ueno@riec.tohoku.ac.jp, homma@riec.tohoku.ac.jp

Keywords: Memory encryption · Authentication Tree · Latency · Mode of Operations · SGX

Abstract. Memory encryption with an authentication tree has received significant attentions due to the increasing threats of active attacks and the widespread use of non-volatile memories. It is also gradually deployed to real-world systems, as shown by SGX available in Intel processors. The topic of memory encryption has been recently extensively studied, most actively from the viewpoint of system architecture. In this paper, we study the topic from the viewpoint of provable secure symmetric-key designs, with a primal focus on latency which is an important criterion for memory. A progress in such a direction can be observed in the memory encryption scheme inside SGX (SGX integrity tree or SIT). It uses dedicated, low-latency symmetric-key components, *i.e.*, a message authentication code (MAC) and an authenticated encryption (AE) scheme based on AES-GCM. SIT has an excellent latency, however, it has a scalability issue for its on-chip memory size. By carefully examining the required behavior of MAC and AE schemes and their interactions in the tree operations, we develop a new memory encryption scheme called ELM. It consists of fully-parallelizable, low-latency MAC and AE schemes and utilizes an incremental property of the MAC. Our AE scheme is similar to OCB, however it improves OCB in terms of decryption latency. To showcase the effectiveness, we consider instantiations of ELM using the same cryptographic cores as SIT, and show that ELM has significantly lower latency than SIT for large memories. We also conducted preliminary hardware implementations to show that the total implementation size is comparable to SIT.

1 Introduction

Cryptographic protection of memory, or more generally storage data, is widely deployed in modern systems. One typical method of protection is sector-wise encryption, such as XTS [Dwo10]. A sector-wise encryption scheme encrypts each memory sector in an independent and deterministic manner, keeping the secret key in a secure on-chip area. This prevents passive off-line attacks that try to extract the data from the storage devices, such as the Cold Boot Attack [HSH$^+$08]. However, it does not offer sufficient protection against active on-line attacks, as there is no way to detect forgeries. Most notably, replay cannot be detected. If we independently encrypt each sector using a nonce-based authenticated encryption (AE) and store all the nonces in the secure on-chip area, it would provide a strong security guarantee against such an adversary. However, this would also incur a linear increase of the on-chip area. This is usually impractical because the on-chip area is much more expensive than the main (off-chip) memory.

A well-known classical solution to this problem is to use an authentication tree, also known as a Merkle Hash Tree [Mer88]. By involving any unit memory data in the tree computation and storing the root hash value in the on-chip area, the authenticity against active attackers can be guaranteed. Instead of a cryptographic hash function, we can use a message authentication code (MAC) to build an authentication tree. The classical Merkle tree and its (possibly MAC-based) improvements, such as PAT [HJ06] and Bonsai Tree [RCPS07], provide an authenticity of the whole memory with a constant on-chip memory overhead, at the cost of a logarithmic computation overhead for read and write operations. Confidentiality of the memory can be achieved by an additional symmetric-key encryption mechanism, as was done by TEC-tree [ECL$^+$07]. Due to the increasing threat of active attacks, authentication trees, often with a confidentiality mechanism, are gradually being deployed in real-world memory/storage systems. One prominent example is Intel's SGX [Gue16b, Gue16a], which adopts a variant of PAT with a dedicated AES-based MAC and AE schemes similar to GMAC and GCM [Dwo07]. The widespread use of non-volatile memory also pushes the need for such protections.

Latency of Memory Protection. Latency is a very important criterion for the aforementioned tree-based memory protection schemes. The Merkle tree can reduce its latency by utilizing parallelizability, but this is only done for verification of the current data. This is usually associated with the memory read operation. When one wants to change the data and re-computes the corresponding authentication value, which is associated with memory write operations, the Merkle tree needs to update all hash values on the path from the leaf (data) to the root in a serial manner. PAT is the current state-of-the-art in this respect, as it is parallelizable for both read and write operations by means of a clever use of nonce-based MAC functions.

Since the introduction of classical Merkle tree, many innovative designs have been proposed in the context of tree-based memory protection [HJ06, YEP$^+$06, RCPS07, ECL$^+$07, TSB18, SNR$^+$18] most actively from the computer architecture perspective. The primary focus of these proposals is their data structure, such as the parameters/structures of integrity trees [RCPS07, TSB18] and counter/nonce representations [YEP$^+$06, SNR$^+$18] that are suitable to the considered architecture. In these proposals, cryptographic components are often considered as black boxes and sometimes instantiated by picking a standard (*e.g.*, GCM in [YEP$^+$06]). A notable exception is the aforementioned scheme used by SGX, which we call the SGX integrity tree (SIT). It develops dedicated AE and MAC schemes based on AES-GCM, with particular attention to latency in mind. It is quite efficient and enables a very low-latency read/write operation on the given tree structure that covers up to 96 Mbyte of memory on an x86 platform. Moreover, as an important subsystem of SGX, it is also quite widely deployed in practice.

Our Contributions. In light of the literature on memory protection schemes thus far, we feel there is a lack of thorough study from the viewpoint of symmetric-key cryptographic design: that is, designing cryptographic components (*e.g.*, modes of block cipher operations) so that they fit well when used in the authentication tree, rather than adopting existing efficient stand-alone modes and using them in a black-box manner. SIT shows potential in this regard and is promising as an excellent low-latency system, but its on-chip data size is linear to the unit data size. This poses a limitation on the amount of covered memory sizes and hence is not scalable. In fact, VAULT [TSB18] and Morphable Counter [SNR$^+$18] are two recent proposals that aim at extending the protected memory size by SIT and improving the performance, mainly from the system architecture viewpoint, using similar symmetric-key components as SIT[3].

After taking a closer look at the interactions between the cryptographic components and the tree operations, we propose a new memory protection scheme dubbed ELM[4], which enables a significantly low latency for a large memory. It achieves on-chip and off-chip memory overheads comparable to existing schemes. ELM combines several techniques from the mode of operations. Specifically, we show the idea of *incremental MAC* introduced by Bellare *et al.* [BGG94, BM97] works quite effectively when the number of branches of the tree is high, which is common for the recent proposals that cover a large memory size, *e.g.*, [TSB18, SNR$^+$18]. By using an incremental MAC at the internal nodes, ELM significantly reduces the write latency without harming the read latency. While our MAC scheme is a variant of the classical XOR-MAC [BGR95], it is carefully designed to optimize the latency, number of primitive calls, parallelizability, and security.

As a key component of ELM, we develop a new low-latency variant of OCB [RBBK01, Rog04, KR11] for AE. OCB is already quite good in terms of latency and parallelizability – better than GCM and other popular schemes, as OCB does not need an additional authentication function. However, the decryption latency of OCB is not sufficiently small due to its structure. By changing the structure, our AE mode has a smaller latency than the original for decryption, while retaining the other main features. In particular,

[3] Vault adopts an OCB-like, AES-based AE instead of GCM without provable security analysis. To our understanding, it needs a very strong related-key security assumption on AES.

[4] Elms are deciduous trees that grow quickly, and we also mean "Encryption for Large Memory" (ELM) by it.

when it is viewed as a mode of a tweakable block cipher (TBC) [LRW02], its latency is *optimally small* for both encryption and decryption. We call it Flat-OCB for its "flat" structure. As well as the original OCB, we proved that our AE is provable secure under the standard cryptographic assumption on AES, *i.e.*, the strong pseudorandomness. Each technique itself is not ultimately novel. However, we show how to combine them in an optimal manner to reduce latency and computation (*e.g.*, by shaving the redundant computations in the update of incremental MACs), which is, to the best of our knowledge, the first time this has been done in the field of tree-based memory encryption.

Our proposal is generic in principle, and the core idea can be instantiated by any block cipher or TBC. To showcase the effectiveness of our proposal, we specify concrete schemes, named ELM1 and ELM2, using the same components as SIT, namely AES-128 and a full 64-bit field multiplier[5]. We compare them with (a generalized variant of) SIT for various memory sizes and tree parameters under a certain practical implementation setting. Our results show that ELM1 and ELM2 have a smaller latency than SIT for most of the cases we see[6]. In particular, when the memory size gets larger, the difference becomes significant. We also conducted preliminary ASIC implementations, and show that the total implementation size is comparable to that of SIT. In addition, we discuss the optimization of hardware implementations for our proposal depending on the system constraints.

2 Preliminaries

2.1 Notation

For a natural number $n \in \mathbb{N}$, $\{0,1\}^n$ denotes the set of n-bit strings. For binary strings A and B, $A \parallel B$ or AB denotes the concatenation of A and B. The bit length of A is denoted by $|A|$, and $|A|_n \coloneqq \lceil |A|/n \rceil$. Dividing a string A into blocks of n bits is denoted by $A[1] \parallel \cdots \parallel A[m] \xleftarrow{n} A$, where $m = |A|_n$ and $|A[i]| = n$, $|A[m]| \le n$ for $1 \le i \le m-1$. For $t \in \mathbb{N}$ and $t \le |A|$, $\mathtt{msb}_t(A)$ ($\mathtt{lsb}_t$) denotes the first (last) t bits of A. A sequence of i zeros is written as 0^i. For sets $\mathbb{E}$ and $\mathbb{E}'$, we write $\mathbb{E} \overset{\cup}{\leftarrow} \mathbb{E}'$ as shorthand for $\mathbb{E} \leftarrow \mathbb{E} \cup \mathbb{E}'$. When the element K is uniformly and randomly chosen from the set $\mathcal{K}$, it is denoted by $K \overset{\$}{\leftarrow} \mathcal{K}$. For a function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ with the key space $\mathcal{K}$, $F(K, \cdot)$ may be written as $F_K(\cdot)$.

Computation on Galois Field. Let $\mathrm{GF}(2^n)$ be a finite field of size 2^n, where the characteristic is 2 and the extension degree is $n \in \mathbb{N}$. We focus on the case where $n = 128$. Following [Rog04, IK03], we use the lexicographically first polynomial for defining the field and thus $\mathbb{F}_{2^{128}} \coloneqq \mathbb{F}_2[x]/(x^{128} + x^7 + x^2 + x + 1)$ and we obtain $\mathrm{GF}(2^n) = \langle x \rangle$. We regard an element of $\mathrm{GF}(2^n)$ as a polynomial of x. For $\forall a \in \{0,1\}^n$, we also regard it as a coefficient vector of an element in $\mathrm{GF}(2^n)$. Thus, the primitive root x is interpreted as 2 in the decimal representation. For $a \in \mathrm{GF}(2^n)$, let $2a$ denote a multiplication by x and a, which is also called doubling [Rog04]. Similarly, let $3a$ denote $2a \oplus a$. In $\mathrm{GF}(2^n)$, $2a \coloneqq (a \ll 1)$ if $\mathtt{msb}_1(a) = 0$ and $2a \coloneqq (a \ll 1) \oplus (0^{120}10^41^3)$ if $\mathtt{msb}_1(a) = 1$, where $(a \ll 1)$ is the left-shift of one bit. For $c \in \mathbb{N}$, we can compute $2^c a$ by doubling a for c times.

2.2 (Tweakable) Block Cipher

Let $\mathcal{K}$ and $\mathcal{M}$ be the set of keys and messages, respectively. Let $\mathcal{T}$ be the set of tweaks, where a tweak is a public parameter. A tweakable block cipher (TBC) [LRW02] is a function $\widetilde{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \to \mathcal{M}$ s.t. $\widetilde{E}(K, T, \cdot)$ is a permutation on $\mathcal{M}$ for $\forall (K, T) \in \mathcal{K} \times \mathcal{T}$. It is also denoted by $\widetilde{E}^T_K$, $\widetilde{E}^T$, or $\widetilde{E}$, where $K \in \mathcal{K}$ and $T \in \mathcal{T}$. If $\mathcal{T}$ is singleton (and we thus omit it from the notation) it means a plain block cipher. Namely, a block cipher E is defined as $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ s.t. $E(K, \cdot)$ is a permutation on $\mathcal{M}$ for $\forall K \in \mathcal{K}$ and is also denoted by E_K or E. A TBC can be built on a block cipher using various modes of operation [LRW02, Rog04].

[5] We also use a 128-bit multiplier, but with a very small input size.

[6] This holds true even when SIT adopts a part of our idea of using incremental MAC. See Section 6.3.

Security Notion. Let $\mathrm{Perm}(n)$ denote the set of all permutations on $\{0,1\}^n$. An n-bit tweakable permutation of t-bit tweak is a function $\pi : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$ s.t. for $\forall T \in \{0,1\}^t$, $\pi(T, \cdot) \in \mathrm{Perm}(n)$. The set of all n-bit tweakable permutations with t-bit tweak is denoted by $\mathrm{TPerm}(t,n)$. Let P s.t. $\mathsf{P} \xleftarrow{\$} \mathrm{Perm}(n)$ be a uniform random permutation (URP) and $\widetilde{\mathsf{P}}$ s.t. $\widetilde{\mathsf{P}} \xleftarrow{\$} \mathrm{TPerm}(t,n)$ be a tweakable URP (TURP). A block cipher E or a TBC $\widetilde{E}$ is said to be secure if it is computationally hard to distinguish from the ideal primitive with oracle access. More precisely, let $\mathcal{A}$ be an adversary who (possibly adaptively) queries an oracle O and subsequently outputs a bit. We write $\Pr[\mathcal{A}^O \to 1]$ to denote the probability that this bit is 1. We define the advantage of $\mathcal{A}$ against TBC $\widetilde{E}$ as follows:

$$\mathbf{Adv}^{\mathrm{tprp}}_{\widetilde{E}}(\mathcal{A}) := |\Pr[\mathcal{A}^{\widetilde{E}} \to 1] - \Pr[\mathcal{A}^{\widetilde{\mathsf{P}}} \to 1]|,$$
$$\mathbf{Adv}^{\mathrm{tsprp}}_{\widetilde{E}}(\mathcal{A}^{\pm}) := |\Pr[(\mathcal{A}^{\pm})^{\widetilde{E},\widetilde{E}^{-1}} \to 1] - \Pr[(\mathcal{A}^{\pm})^{\widetilde{\mathsf{P}},\widetilde{\mathsf{P}}^{-1}} \to 1]|,$$

where the first notion is for adversary with encryption oracle (*i.e.*, chosen-plaintext queries), and the second is for adversary with encryption and decryption oracles (*i.e.*, chosen-ciphertext queries). When the advantage is sufficiently small, $\widetilde{E}$ is said to be secure against the underlying adversary.

2.3 Message Authentication Code

Message authentication code (MAC) is a symmetric-key cryptosystem to ensure the integrity of a message. Throughout the paper, we consider *nonce-based* MAC[7]. It takes a nonce (a value never repeats when used for tag generation) together with a message. For the key space $\mathcal{K}$, the nonce space $\mathcal{N}$, the message space $\mathcal{M}$, and the tag space $\mathcal{T}$, a nonce-based MAC scheme MAC consists of two functions; the tagging function $\mathsf{MAC}.\mathcal{T} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \to \mathcal{T}$ and the verification function $\mathsf{MAC}.\mathcal{V} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \times \mathcal{T} \to \{\top, \bot\}$. A tag for the message $M \in \mathcal{M}$ and the nonce $N \in \mathcal{N}$ and the key $K \in \mathcal{K}$ is $T = \mathsf{MAC}.\mathcal{T}(K, N, M)$. The tuple (N, M, T) is considered to be authentic if $\mathsf{MAC}.\mathcal{T}(K, N, M, T) = \top$, and otherwise it is rejected.

Security Notion. Let $\mathcal{A}$ be the adversary against MAC described above. The security of MAC is defined as the probability that $\mathcal{A}$ creates a successful forgery by accessing the tagging oracle ($\mathsf{MAC}.\mathcal{T}_K$) and the verification oracle ($\mathsf{MAC}.\mathcal{V}_K$). The security measure is $\mathbf{Adv}^{\mathsf{mac}}_{\mathsf{MAC}}(\mathcal{A}) := \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathsf{MAC}.\mathcal{T}_K, \mathsf{MAC}.\mathcal{V}_K} \text{forges}]$, which means the probability of a successful forgery. That is, $\mathcal{A}$ receives $\top$ from $\mathsf{MAC}.\mathcal{V}_K$ by querying (N', M', T') while (N', M') has never been queried to $\mathsf{MAC}.\mathcal{T}_K$. Here, $\mathcal{A}$ is assumed to be nonce-respecting, that is, the nonces in the tagging queries are distinct. The nonces in the verification queries have no restriction, and $\mathcal{A}$ can repeat nonce or reuse a nonce that was used by a tagging query.

2.4 Authenticated Encryption

Authenticated encryption (AE) [BN00] is used to ensure privacy and authenticity of input data simultaneously. As well as MAC, we consider *nonce-based* AE in this paper. For the key space $\mathcal{K}$, the nonce space $\mathcal{N}$, the message and ciphertext space $\mathcal{M}$, and the tag space $\mathcal{T}$, a nonce-based AE scheme AE consists of two functions; the encryption function $\mathsf{AE}.\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \to \mathcal{M} \times \mathcal{T}$ and the decryption function $\mathsf{AE}.\mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \times \mathcal{T} \to \mathcal{M} \times \bot$. A ciphertext $C \in \mathcal{M}$ and a tag $T \in \mathcal{T}$ for the key $K \in \mathcal{K}$, the nonce $N \in \mathcal{N}$, and the message $M \in \mathcal{M}$ are derived as $(C, T) = \mathsf{AE}.\mathcal{E}(K, N, M)$. The tuple (N, C, T) is considered to be authentic if $\mathsf{AE}.\mathcal{D}(K, N, C, T)$ returns the message $M \in \mathcal{M}$ and $M \neq \bot$, and otherwise it is rejected.

It is possible to extend AE so that it also accepts *associated data* [Rog02], an information that is not encrypted but authenticated, though we do not need it in this paper.

[7] It is because a nonce-based MAC is generally easier to construct than a general (non-nonce-based) MAC to construct an efficient MAC. The same applies to a nonce-based AE.

Security Notion. The security of AE is evaluated by two criteria, privacy and authenticity advantages. The privacy advantage is the probability that the adversary successfully distinguishes the encryption function of AE from the random oracle $\$(*, *)$. For any query (N, M), if $(C, T) \leftarrow \mathsf{AE}.\mathcal{E}(K, N, M)$, $\$(N, M)$ returns random bits of length $|C| + |T|$. Thus, $\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{AE}}(\mathcal{A}) := |\Pr[\mathcal{A}^{\mathsf{AE}.\mathcal{E}} \to 1] - \Pr[\mathcal{A}^{\$} \to 1]|$. The authenticity advantage is the probability that the adversary creates a successful forgery by accessing encryption function and decryption function. It is defined as $\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{AE}}(\mathcal{A}) := \Pr[\mathcal{A}^{\mathsf{AE}.\mathcal{E},\mathsf{AE}.\mathcal{D}}$ forges], which means the probability that $\mathcal{A}$ receives $M' \neq \bot$ from $\mathsf{AE}.\mathcal{D}$ by querying (N', C', T') while (N', M') has never been queried to $\mathsf{AE}.\mathcal{E}$.

For both advantages, we assume the adversary is nonce-respecting in encryption queries. For authenticity however, there is no restriction on nonce in the decryption queries, that is, $\mathcal{A}$ may repeat a nonce or reuse a nonce that was used in an encryption query.

2.5 Authentication Tree for Memory Protection

We assume two regions in storage memory: on-chip and off-chip areas. On-chip area is assumed to be secure in which the adversary cannot eavesdrop or tamper the stored data. Off-chip area can be attacked by the adversary who may perform eavesdropping (getting information of plaintext from ciphertext), tampering (modify the ciphertext without being detected), and replay (replacing the ciphertext with an old legitimate one). As mentioned in the introduction, tampering can be detected by simply applying a MAC to each data unit and storing the nonce and tag off-chip. If we use a nonce-based AE scheme instead, it also prevents eavesdropping. However, these means are not sufficient to protect from replay attacks since the adversary can perform a replay on the (nonce,ciphertext,tag) tuple. Moreover, since on-chip area is generally much more expensive than off-chip area, it is desirable to thwart all of these attacks with a small amount on-chip area as possible.

To address the problem, a number of memory protection tree schemes have been proposed [Mer88, RCPS07, HJ06, UWM19, TSB18, SNR+18]. The classical Merkle hash tree [Mer88] associates each memory data chunk stored off-chip to a leaf node of a tree. The hash values of all intermediate and leaf nodes are stored off-chip, and only that of the root node is stored on-chip. The integrity of a leaf node (data) can be verified by recursively computing the corresponding hash values from the leaf to the root.

A similar scheme can be considered by using MACs instead of hash functions by storing the secret key on-chip, and among such schemes, we focus on PAT (Parallelizable Authentication Tree) proposed by Hall and Jutla [HJ06] for its parallelizability of both verify and update operations. It assigns a nonce to each node and stores the nonce associated with the root node in the on-chip area. Here, nonces need to be distinct from each other, and have one-time property to prevent replay attacks. To construct parallel scheme, PAT employs a MAC to compute a tag by taking the nonce assigned to own node and nonces in children nodes[8].

In this paper, we hereafter use the term *authentication tree* to refer to the memory protection scheme using the tree construction. Note that we suppose the authentication tree also encrypts data associated with leaf nodes. We introduce a generic construction of authentication tree $\mathsf{PAT2}$ (Fig. 1). It is mostly identical to PAT, however it achieves confidentiality of memory by applying an AE scheme to the leaf nodes[9] and it splits any nonce of PAT associated to a node into two values, an address and a local counter. The former is the memory address of the node, and the latter is a counter exclusively assigned to the node.

Let us briefly describe how $\mathsf{PAT2}$ of Fig. 1 works. Each nonce N_i assigned to node i consists of the address addr_i and the local counter ctr_i, which is initialized to 0 for all nodes. Memory data is split into 4 units, M_3 to M_6. After initialization, the tree keeps N_i, T_i for $i = 1, \ldots, 6$, and C_j for $j = 3, \ldots, 6$ at the off-chip area, and N_0 at the on-chip area. When verifying a data, say M_3, we check if (1) $\mathrm{AE}.\mathcal{D}_K(N_3, C_3, T_3)$ is authentic (*i.e.*, not returning $\bot$) and (2) $\mathrm{MAC}.\mathcal{V}_{K'}(N_1, \mathsf{ctr}_3 \,\|\, \mathsf{ctr}_4, T_1) = \top$ and (3) $\mathrm{MAC}.\mathcal{V}_{K'}(N_0, \mathsf{ctr}_1 \,\|\, \mathsf{ctr}_2, T_0) = \top$. If all hold M_3 is considered to be authentic and the corresponding local counters (ctr_0, ctr_1 and ctr_3) are incremented. When updating M_3, we first perform the above verification procedure, update the counters, and

[8] To be more precise, [HJ06] proposes to use a general deterministic MAC with input being prepended by a nonce, which is a typical way to convert a deterministic MAC into a nonce-based one.

[9] In fact, An ePrint version of PAT paper [HJ02] specifies a combination of MAC and AE schemes for confidentiality of leaf data.

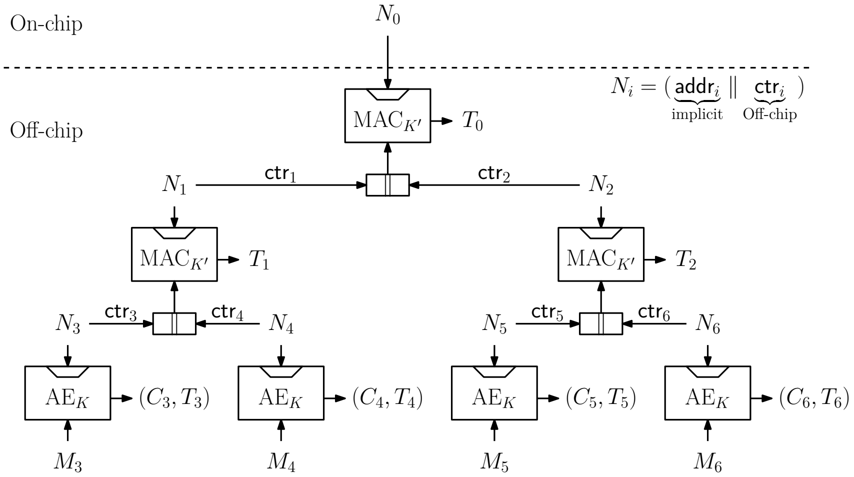


Fig. 1: An example of PAT2 with tree depth 2 and number of branches 2. A trapezoid in a MAC or an AE box denotes the nonce input, and a box with $\|$ denotes concatenation. For $0 \leq i \leq 2$, $T_i = \mathsf{MAC}_{K'}(N_i, \mathsf{ctr}_{2i+1} \,\|\, \mathsf{ctr}_{2i+2})$. For $3 \leq j \leq 6$, $(C_j, T_j) = \mathsf{AE}.\mathcal{E}_K(N_j, M_j)$. Only ctr_0 is stored in the on-chip area.

then renew (N_3, C_3, T_3), N_1, T_1 and T_0. Observe that the steps in the verification and update procedures are independent and thus parallelizable. This is a crucial advantage of PAT/PAT2 over the classical hash tree which only allows parallel verification. The nonce format guarantees distinctness across different nodes, and allows to reduce the MAC input and off-chip overhead from the original PAT. Since an address is anyway given from the outer legitimate system, it does not need to be explicitly stored. To the best of our knowledge, this technique was first proposed by [RCPS07].

In fact, by specifying the parameters (*e.g.*, the depth and the branch number of the tree and the format of nonce) and the underlying MAC and AE schemes, the resulting scheme is mostly identical to SIT. Therefore, PAT2 can be seen as an abstraction of SIT. We consider PAT2 as our baseline scheme for its simple structure and efficiency, and present our scheme based on it (Section 4).

To the best of our knowledge, the provable security of PAT2 have not been shown in literature. As described above, many memory encryption schemes have been proposed, but there are few papers which shows provable security of proposed schemes. Whereas PAT paper defines the security notion of integrity tree (*i.e.*, tree-based memory protection scheme, but not encrypting memory) and proves the security of PAT, PAT has slightly different tree construction from PAT2 and there is no description about privacy of plaintext associated with leaf nodes. In Section 5, we define security notions (privacy and unforgeability) for authentication trees and prove the security of PAT2 in each notion. The analysis is not surprising, but to our knowledge we cannot find such a formal treatment (in particular for the combination of MAC and AE to guarantee privacy and unforgeability) in literature.

Other Schemes. In addition to the above schemes, a number of authentication tree schemes that better handle the various criteria (except for latency) have been proposed. TEC-tree [ECL$^+$07] provides confidentiality by encrypting data stored in all nodes. MAES [UWM19] is an authentication tree providing security against differential power analysis attacks. VAULT [TSB18] and Morphable Counter [SNR$^+$18] reduce the overhead of off-chip memory and are suitable for protecting large memory (*e.g.*, larger than the giga byte order); however there is a tradeoff with the average latency because their counters are compressed.

Algorithm 1 PXOR-MAC.$\mathcal{T}_{K,K'}(N, M)$

```
$M[1] \parallel \cdots \parallel M[m] \xleftarrow{n} M$
$L \leftarrow E_K(0^n)$, $T \leftarrow 0^\tau$
for $1 \leq i \leq m$ do
    $T \leftarrow T \oplus \mathtt{msb}_\tau(E_K(M[i] \oplus K' \cdot i))$
end for
$T \leftarrow T \oplus \mathtt{msb}_\tau(E_K(N \oplus K' \cdot m \oplus L))$
return $T$
```

Algorithm 2 PXOR-MAC.$\mathcal{V}_{K,K'}(N, M, T)$

```
$T' \leftarrow$ PXOR-MAC.$\mathcal{T}_{E_K,K'}(N, M)$
if $T = T'$ then
    return $\top$
else
    return $\bot$
end if
```

Algorithm 3 PXOR-MAC.$\mathcal{U}_{K,K'}(N_{\text{old}}, M_{\text{old}}, T_{\text{old}}, N_{\text{new}}, M_{\text{new}})$

```
$L \leftarrow E_K(0^n)$, $T_{\text{new}} \leftarrow T_{\text{old}}$
$M_{\text{old}}[1] \parallel \cdots \parallel M_{\text{old}}[m] \xleftarrow{n} M_{\text{old}}$, $M_{\text{new}}[1] \parallel \cdots \parallel M_{\text{new}}[m] \xleftarrow{n} M_{\text{new}}$
for $1 \leq i \leq m$ do
    if $M_{\text{old}}[i] \neq M_{\text{new}}[i]$ then
        $T_{\text{new}} \leftarrow T_{\text{new}} \oplus \mathtt{msb}_\tau(E_K(M_{\text{old}}[i] \oplus K' \cdot i) \oplus E_K(M_{\text{new}}[i] \oplus K' \cdot i))$
    end if
end for
if $N_{\text{old}} \neq N_{\text{new}}$ then
    $T_{\text{new}} \leftarrow T_{\text{new}} \oplus \mathtt{msb}_\tau(E_K(N_{\text{old}} \oplus K' \cdot m \oplus L) \oplus E_K(N_{\text{new}} \oplus K' \cdot m \oplus L))$
end if
return $T_{\text{new}}$
```

3 Components of ELM

To achieve low latency operation, we designed dedicated MAC and AE schemes. Our MAC scheme, which we call PXOR-MAC, is a simple combination of nonce-based XOR-MAC [BGR95] and PHASH, a message hashing function of PMAC [BR02, Rog04]. For AE, we propose a new mode named Flat-OCB based on OCB. We show more details in the following.

3.1 Incremental MAC

Specification. Algs. 1 and 2 show the tagging function and the verification function of PXOR-MAC. Here, the block cipher E has n-bit block. The second key K' is n bits and independent of K. The length of the nonce N and the tag T are n bits and τ bits, respectively. Note that we exclude the case of partial block (*i.e.*, $|M| \mod n = 0$ always holds.) for simplicity. As we assumed $n = 128$, this is reasonable for the typical use case of authentication tree schemes. PXOR-MAC computes a tag as the sum of encrypted plaintext blocks and the encrypted nonce, as depicted in Fig. 2. An input mask to E_K is derived from a multiplication of K' and the block index over $\mathrm{GF}(2^n)$, and $L = E_K(0^n)$.

Properties. Since L can be computed in advance, the latency of tag computation is essentially a sum of latencies of 128-bit multiplication ($K' \cdot i$ for the block index i) and one call of E_K. The cost of 128-bit multiplication can be large if i has large variations, however m is not too large in practice, even when the total memory size is huge. Typically, m is upperbounded by the number of branches, for example, at most 2^7 according to [SNR$^+$18]. Therefore, using a Gray code, the hardware implementation is much more efficient compared to implementing a full 128-bit multiplier (see Section 6). Consequently, the latency of mask computation becomes negligible. In this setting, PXOR-MAC has optimal latency of one block cipher call for tagging and verification functions thanks to the full parallelizability of block cipher calls.

In addition, PXOR-MAC is an incremental MAC [BGR95], which allows an efficient tag computation when a message is changed at a small number of blocks. To be more concrete, when a block of a message is changed

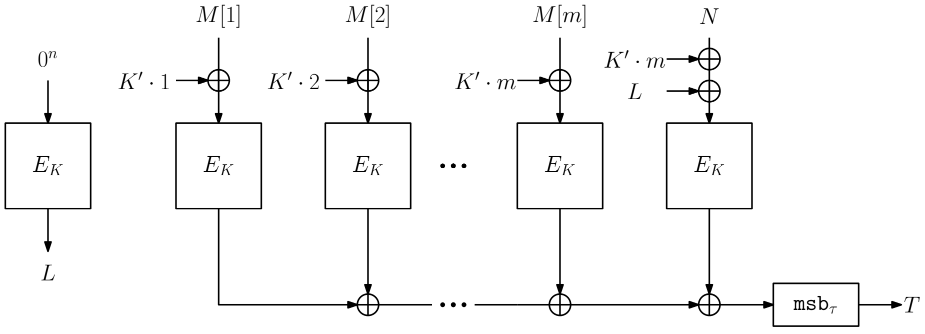


Fig. 2: PXOR-MAC.

together with a new nonce (since a nonce-based MAC renews its nonce for each tag generation), the new tag can be obtained by encrypting the corresponding blocks (*i.e.*, XOR of the message block and its mask value) for both old and new ones, and taking an XOR of them and the old tag. We show the general update function of PXOR-MAC in Alg. 3. It takes old nonce N_{old}, old plaintext M_{old}, old tag T_{old}, new nonce N_{new}, and new plaintext M_{new}. It outputs new tag T_{new} such that $T_{\text{new}} = \mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K,K'}(N_{\text{new}}, M_{\text{new}})$ holds. For simplicity, Alg. 3 assumes that M_{old} and M_{new} have the same number of blocks, m. In this case, when the nonce and one-block plaintext are changed, $\mathsf{PXOR\text{-}MAC}.\mathcal{U}$ needs only four block cipher calls except for mask derivation regardless of m, while ordinary block cipher-based MACs have to invoke block cipher at least m times by invoking their tagging functions.

Notes on Incremental Property. Our PXOR-MAC corresponds to the incremental MAC for *replace* operation with *basic* security [BGG94]. We emphasize that the arguments of the update function defined at [BGG94] is different from those of Alg. 3. In detail, the update function in [BGG94] takes the set of block indices to replace, and the contents of new and old blocks in addition to the old tag T_{old}. For notational convenience we adopted our presentation of Alg. 3, however we used the standard form of [BGG94] in our implementations for efficiency. In addition, the basic security means that $T_{\text{old}} = \mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K,K'}(N_{\text{old}}, M_{\text{old}})$ must hold for any $(N_{\text{old}}, M_{\text{old}}, T_{\text{old}})$ in an input to the update function to guarantee the correctness. This implies that the update function cannot be queried by the adversary. Bellare *et al.* [BGG94] also defined a stronger, tamper-proof security, where the adversary can arbitrary query the update oracle. This is a crucially different security notion, as the adversary may feed an unauthentic tuple $(N_{\text{old}}, M_{\text{old}}, T_{\text{old}})$ to the update oracle. Fortunately, an incremental MAC with basic security suffices for our purpose (see Section 4.3).

Security. The security bound of PXOR-MAC is shown below. We assume the underlying block cipher is an n-bit URP P. It is an information-theoretic idealization. The computational counterpart, where the underlying block cipher is instantiated by a practically secure block cipher such as AES, is derived from our bound. Since this is fairly standard [BDJR97], we omitted it here.

Theorem 1. *The MAC advantage of* PXOR-MAC *is*

$$\mathbf{Adv}^{\mathtt{mac}}_{\mathsf{PXOR\text{-}MAC}_{\mathsf{P}}}(\mathcal{A}) \leq \frac{2q_v}{2^\tau} + \frac{4.5\sigma^2_{\mathtt{mac}}}{2^n},$$

where $\mathcal{A}$ is the nonce-respecting adversary against PXOR-MAC*, $\sigma_{\mathtt{mac}}$ is the total number of accesses to* P *invoked by the queries such that $\sigma_{\mathtt{mac}} \leq 2^{n-1}$, and q_v is the number of queries to the verification oracle.*

Proof. First, we observe that PXOR-MAC can be interpreted as a TBC-based MAC, PXOR-MAC-TBC, defined in Algs. 4 and 5. If the TBC used in PXOR-MAC-TBC is specified as $\widetilde{E}^{0^n,i,j}_{K,K'}(M) = E_K(M \oplus K' \cdot i \oplus j \cdot E_K(0^n))$, $\mathsf{PXOR\text{-}MAC\text{-}TBC}_{\widetilde{E}}$ is equivalent to $\mathsf{PXOR\text{-}MAC}_{K,K'}$. Thus, we have

$$\mathbf{Adv}^{\mathtt{mac}}_{\mathsf{PXOR\text{-}MAC}_{\mathsf{P}}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathtt{mac}}_{\mathsf{PXOR\text{-}MAC\text{-}TBC}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) + \mathbf{Adv}^{\text{tprp}}_{\widetilde{\mathsf{E}}}(\mathcal{B}), \tag{1}$$

where P is an n-bit URP, $\widetilde{\mathsf{P}}$ is an n-bit TURP having the same tweak space as $\widetilde{E}$, and $\widetilde{\mathsf{E}}$ is a TBC involving P and an independent key K', defined as $\widetilde{\mathsf{E}}^{0^n,i,j}(M) = \mathsf{P}(M \oplus K' \cdot i \oplus j \cdot \mathsf{P}(0^n))$. Also, $\mathcal{B}$ is the adversary against $\widetilde{\mathsf{E}}$ querying the encryption oracle. In what follows, we evaluate each term of the right side of (1) in turn. Recall that we assume $|M| \pmod n = 0$ for plaintext M.

Algorithm 4 $\mathsf{PXOR\text{-}MAC\text{-}TBC}.\mathcal{T}_{\widetilde{E}}(N, M)$

```
$M[1] \| \cdots \| M[m] \xleftarrow{n} M$
$T \leftarrow 0^\tau$
for $1 \le i \le m$ do
    $T \leftarrow T \oplus \mathtt{msb}_\tau(\widetilde{E}^{0^n,i,0}(M[i]))$
end for
$T \leftarrow T \oplus \mathtt{msb}_\tau(\widetilde{E}^{0^n,m,1}(N))$
return $T$
```

Algorithm 5 $\mathsf{PXOR\text{-}MAC\text{-}TBC}.\mathcal{V}_{\widetilde{E}}(N, M, T)$

```
$M[1] \| \cdots \| M[m] \xleftarrow{n} M$
$T' \leftarrow \mathsf{PXOR\text{-}MAC\text{-}TBC}.\mathcal{T}_{\widetilde{E}}(N, M)$
if $T = T'$ then
    return $\top$
else
    return $\bot$
end if
```

Analysis of the First Term. We start with the case $q_v = 1$. Without loss of generality, we can assume that the adversary performs a verification query after all q_t tagging queries. Let $Z = \{(N_1, M_1, T_1), \ldots, (N_{q_t}, M_{q_t}, T_{q_t})\}$ be the transcript obtained by tagging queries, and let (N', M', T') be the verification query. Let T^* be the valid tag corresponding to (N', M'). We also suppose $|M'|_n \coloneqq m'$. Seeing Z as a random variable, we obtain $\mathbf{Adv}^{\mathsf{mac}}_{\mathsf{PXOR\text{-}MAC\text{-}TBC}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) = \sum_z \Pr[T' = T^* \mid Z = z] \Pr[Z = z]$. In what follows, we evaluate $\Pr[T' = T^* \mid Z = z]$ for the following cases.

1. Let $\forall i \in \{1, \ldots, q_t\}, N' \neq N_i$.
   The TURP $\widetilde{\mathsf{P}}^{0^n,m',1}$, which encrypts nonce N', can be invoked at most q_t times in tagging queries. However, $\widetilde{\mathsf{P}}^{0^n,m',1}(N')$ is a new random value for the adversary. Thus, supposing that $q_t \le 2^{n-1}$, we obtain $\Pr[T' = T^* \mid Z = z] \le 2^{n-\tau}/(2^n - q_t) \le 2/2^\tau$.
2. Let $\exists \alpha \in \{1, \ldots, q_t\}, N' = N_\alpha$ and $m' \neq |M_\alpha|_n$.
   The TURP $\widetilde{\mathsf{P}}^{0^n,m',1}$ encrypting N' has a different tweak from that encrypting N_α in tagging queries. Thus, we can treat this case in the same manner as the previous case, and $\Pr[T' = T^* \mid Z = z] \le 2/2^\tau$ holds.
3. Let $\exists \alpha \in \{1, \ldots, q_e\}, N' = N_\alpha$ and $m' = |M_\alpha|_n$.

   (a) When $M'[i] \neq M_\alpha[i]$ for $\exists! i \in \{1, \ldots, m'\}$ and $M'[j] = M_\alpha[j]$ for $\forall j \in \{1, \ldots, m'\} \setminus \{i\}$, it necessarily holds that $\widetilde{\mathsf{P}}^{0^n,i,0}(M'[i]) \neq \widetilde{\mathsf{P}}^{0^n,i,0}(M_\alpha[i])$ and $\widetilde{\mathsf{P}}^{0^n,j,0}(M'[j]) = \widetilde{\mathsf{P}}^{0^n,j,0}(M_\alpha[j])$. Thus, we obtain $\Pr[T' = T^* \mid Z = z] = \Pr[T' = T_\alpha \oplus \mathtt{msb}_\tau(\widetilde{\mathsf{P}}^{0^n,i,0}(M'[i]) \oplus \widetilde{\mathsf{P}}^{0^n,i,0}(M_\alpha[i])) \mid Z = z] \le 2/2^\tau$.

   (b) When $M'[i] \neq M_\alpha[i]$ and $M'[j] \neq M_\alpha[j]$ for $i, j \in \{1, \ldots, m'\}$, it holds that

   $$T^* = T_\alpha \oplus \mathtt{msb}_\tau(\widetilde{\mathsf{P}}^{0^n,i,0}(M'[i]) \oplus \widetilde{\mathsf{P}}^{0^n,i,0}(M_\alpha[i])) \oplus \mathtt{msb}_\tau(\widetilde{\mathsf{P}}^{0^n,j,0}(M'[j]) \oplus \widetilde{\mathsf{P}}^{0^n,j,0}(M_\alpha[j])) \oplus \delta,$$

   where $\delta = \bigoplus_{k \neq i,j}^{m'} \mathtt{msb}_\tau(\widetilde{\mathsf{P}}^{0^n,k,0}(M'[k]) \oplus \widetilde{\mathsf{P}}^{0^n,k,0}(M_\alpha[k]))$. Thus, we obtain $\Pr[T' = T^* \mid Z = z] \le 2/2^\tau$.

From the above cases, we obtain the following advantage when $q_d = 1$:

$$\mathbf{Adv}^{\mathsf{mac}}_{\mathsf{PXOR\text{-}MAC\text{-}TBC}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) \le \sum_z \max_z \Pr[T' = T^* \mid Z = z] \Pr[Z = z] \le \frac{2}{2^\tau}. \tag{2}$$

Finally, we apply the standard conversion from single to multiple verification queries [BDJR97] and obtain the bound $\mathbf{Adv}^{\mathsf{mac}}_{\mathsf{PXOR\text{-}MAC\text{-}TBC}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) \le q_d\,(2/2^\tau)$ for $q_d \ge 1$.

Analysis of the Second Term. We evaluate $\mathbf{Adv}^{\mathrm{tprp}}_{\widetilde{\mathsf{E}}}(\mathcal{B})$ in (1). We follow the framework proposed in [MM09]. We define the offset function F as follows.

$$F_{K'}((i,j),\mathsf{P}(0^n)) = K' \cdot i \oplus j \cdot \mathsf{P}(0^n),$$

where $i \in \{1,2,\ldots\}$, $j \in \{0,1\}$. Then, $\widetilde{\mathsf{E}}^{0^n,i,j}(M) = \mathsf{P}(M \oplus F_K((i,j),\mathsf{P}(0^n)))$ holds for any (i,j,M). Here, we introduce the following definition and the lemma for offset functions.

Definition 1 (A Simplified Version of Definition 4.1 [MM09]). *Let V be a uniformly random value over $\{0,1\}^n$. We say that a offset function F is $(\varepsilon,\gamma,\rho)$-uniform if F satisfies the following conditions.*

$$\begin{aligned}
&\max_{l\neq l',\delta\in\{0,1\}^n} \Pr[F(l,V)\oplus F(l',V)=\delta] \leq \varepsilon,\\
&\max_{l,\delta\in\{0,1\}^n} \Pr[F(l,V)=\delta] \leq \gamma,\\
&\max_{l,\delta\in\{0,1\}^n} \Pr[F(l,V)\oplus V=\delta] \leq \rho.
\end{aligned}$$

Lemma 1 (A Simplified Version of Theorem 4.1 in [MM09]). *Suppose that $\widetilde{\mathsf{E}}$ uses an $(\varepsilon,\gamma,\rho)$-uniform offset function F. We obtain following evaluation.*

$$\mathbf{Adv}^{\mathrm{tprp}}_{\widetilde{\mathsf{E}}}(\mathcal{B}) \leq q^2\left(2\varepsilon+\gamma+\rho+\frac{1}{2^{n+1}}\right),$$

where q is the number of encryption queries such that $q \leq 2^{n-1}$.

We derive the security bound of $\widetilde{\mathsf{E}}$ by evaluating $(\varepsilon,\gamma,\rho)$ in Definition 1.

Bound of ε. For all $\delta \in \{0,1\}^n$, we bound the probability $\Pr[X(\delta)] \coloneqq \Pr[F((i,j),\mathsf{P}(0^n))\oplus F((i',j'),\mathsf{P}(0^n)) = \delta]$. When $j = j'$, $i \neq i'$ must hold. Thus $\Pr[X(\delta)] = \Pr[K'(i \oplus i') = \delta] \leq 1/2^n$ holds since K' is drawn from $\{0,1\}^n$ uniformly and $i \oplus i' \neq 0$. When $j \neq j'$, $\Pr[X(\delta)] = \Pr[K'(i\oplus i') \oplus \mathsf{P}(0^n) = \delta] \leq 1/2^n$ since K' and $\mathsf{P}(0^n)$ are uniformly random and independent from each other. Thus, $\varepsilon = 1/2^n$ holds.

Bound of γ. Suppose that $j = 0$ holds. For all $\delta \in \{0,1\}^n$, we obtain $\Pr[F((i,0),\mathsf{P}(0^n)) = \delta] = \Pr[K' \cdot i = \delta] \leq 1/2^n$ due to the uniformity of K'. Suppose that $j = 1$ holds. we also obtain $\Pr[F((i,1),\mathsf{P}(0^n)) = \delta] = \Pr[K' \cdot i \oplus \mathsf{P}(0^n) = \delta] \leq 1/2^n$ since K' is uniformly random and independent from P. Thus, $\gamma = 1/2^n$ holds.

Bound of ρ. For all $\delta \in \{0,1\}^n$, we bound the probability $\Pr[Y(\delta)] \coloneqq \Pr[F((i,j),\mathsf{P}(0^n)) \oplus \mathsf{P}(0^n) = \delta]$. When $j = 0$, $\Pr[Y(\delta)] = \Pr[K' \cdot i \oplus \mathsf{P}(0^n) = \delta]$ holds. From the same discussion of γ when $j = 1$, we obtain $\Pr[Y(\delta)] \leq 1/2^n$. When $j = 1$, $\Pr[Y(\delta)] = \Pr[K' \cdot i = \delta]$ holds. From the same discussion of γ when $j = 0$, we obtain $\Pr[Y(\delta)] \leq 1/2^n$. Thus, $\rho = 1/2^n$ holds.

From the above discussions, we obtain

$$\mathbf{Adv}^{\mathrm{tprp}}_{\widetilde{\mathsf{E}}}(\mathcal{B}) \leq \frac{4.5\sigma_{\mathtt{mac}}^2}{2^n}, \tag{3}$$

where $\sigma_{\mathtt{mac}}$ is the number of accesses to P and $\sigma_{\mathtt{mac}} \leq 2^{n-1}$. Combining (1),(2), and (3), we conclude the proof.

3.2 Low-Latency Authenticated Encryption

An AE scheme can be built on a block cipher by a mode of operation. While it is possible to build an AE by a generic composition of a MAC mode and an encryption mode (*e.g.*, Counter mode) [BN00, Kra01, NRS14], OCB is generally faster. It needs m plus a few block cipher calls to process m-block input (while a generic

Algorithm 6 Flat-ΘCB.$\mathcal{E}_{\widetilde{E}}(N, M)$

```
$M[1] \,\|\, \cdots \,\|\, M[m] \xleftarrow{n} M$
$T \leftarrow \mathtt{msb}_\tau(\widetilde{E}_K^{N,0,0}(0^n))$
for $1 \le i \le m-1$ do
    $C[i] \leftarrow \widetilde{E}_K^{N,i,0}(M[i])$
    $T \leftarrow T \oplus \mathtt{msb}_\tau(M[i])$
end for
$C[m] \leftarrow \widetilde{E}_K^{N,m-1,1}(M[m])$
$T \leftarrow T \oplus \mathtt{msb}_\tau(M[m])$
$C \leftarrow C[1] \,\|\, \cdots \,\|\, C[m]$
return $C, T$
```

Algorithm 7 Flat-ΘCB.$\mathcal{D}_{\widetilde{E}}(N, C, T)$

```
$C[1] \,\|\, \cdots \,\|\, C[m] \xleftarrow{n} C$
$T' \leftarrow \mathtt{msb}_\tau(\widetilde{E}_K^{N,0,0}(0^n))$
for $1 \le i \le m-1$ do
    $M[i] \leftarrow \widetilde{D}_K^{N,i,0}(C[i])$
    $T' \leftarrow T' \oplus \mathtt{msb}_\tau(M[i])$
end for
$M[m] \leftarrow \widetilde{D}_K^{N,m-1,1}(C[m])$
$T' \leftarrow T' \oplus \mathtt{msb}_\tau(M[m])$
if $T = T'$ then
    return $M \leftarrow M[1] \,\|\, \cdots \,\|\, M[m]$
else
    return $\perp$
end if
```

Algorithm 8 MASK1(N)

```
return $\mathit{\Delta} \leftarrow \mathsf{AES4}_{K_1,K_2,K_3,K_4}(N)$
```

Algorithm 9 MASK2(N)

```
$N_1 \leftarrow \mathtt{msb}_{n/2}(N)$, $N_2 \leftarrow \mathtt{lsb}_{n/2}(N)$
return $\mathit{\Delta} \leftarrow (N_1 \cdot K_1 \,\|\, N_2 \cdot K_2) \oplus (N_2 \cdot K_3 \,\|\, N_1 \cdot K_4)$
```

composition needs at least $2m$ calls), and these m calls are parallelizable. Thanks to this property, OCB has quite a small latency. However, there is a gap in the latency for encryption and decryption of OCB. Specifically, the encryption of *plaintext checksum* must be done after the main decryption routine. It results in one block cipher call that cannot be computed in parallel, and adds a significant latency compared to the encryption (we detail later). We present a solution to this problem. Because our proposal is essentially an improvement of a TBC-based interpretation of OCB (ΘCB [KR11][10]), we first describe it, which we call Flat-ΘCB. Then we show two block cipher-based instantiations of Flat-ΘCB, denoted by Flat-OCB-f and Flat-OCB-m.

As a related work, Qameleon [ABB$^+$19] is an AE scheme proposed to the ongoing NIST standardization project for lightweight cryptography [NIS19]. It is based on ΘCB using a low-latency TBC QARMA [Ava17] and has the same issue as ΘCB in decryption latency.

Specification. We show Flat-ΘCB in Algs. 6, 7 and Fig. 3. It is an AE mode based on n-bit TBC, $\widetilde{E}$. The nonce N is also assumed to be n bits. As well as the case of MAC, we assume that the case of partial block is excluded for simplicity. The structure of Flat-ΘCB is almost the same as that of ΘCB. The crucial difference is the generation of the tag T. While ΘCB encrypts the checksum $M[1] \oplus M[2] \oplus \ldots \oplus M[m]$ using $\widetilde{E}$ to produce T, ours first encrypts N and take a sum with the checksum.

To build a block cipher-based AE, we instantiate $\widetilde{E}$ with an n-bit block cipher E_K as follows.

$$\widetilde{E}_K^{N,i,j}(M) = E_K(M \oplus \mathit{\Delta} \oplus 2^i \cdot 3^j E_K(0^n)) \oplus \mathit{\Delta} \oplus 2^i \cdot 3^j E_K(0^n), \tag{4}$$

where $i \in \{0, 1, 2, \ldots\}$, $j \in \{0, 1\}$, and the part of mask $\mathit{\Delta}$ is an n-bit value derived from N. We show two derivations of $\mathit{\Delta}$, MASK1 and MASK2, in Alg. 8 and Alg. 9, respectively. MASK1 explicitly requires $n = 128$ (or, more specifically the doubling and tripling yield a safe instantiation of XEX [Rog04]). MASK2 can use any even n. MASK1 computes $\mathit{\Delta}$ by using 4-round AES denoted by AES4 with four independent 128-bit secret keys, as used by the existing MAC and TBC constructions based on AES [MT06, Min07] (this is to utilize 4-round AES's differential property without harming provable security reduction to the entire AES: see below). Thus, it is natural to assume that E in (4) is also AES when MASK1 is used. Here, we assume that 1-round AES is the sequence of operations (`AddRoundKey`, `Subbyte`, `ShiftRows`, `MixColumns`), and four independent

[10] More precisely it is denoted as ΘCB3 in [KR11].

Table 1: Comparison of AE modes. SIT-AE is a GCM-based AE defined by SIT. **Enc Latency** (**Dec latency**) denotes the encryption (decryption) latency in terms of the number of primitive calls. Here, 1 BC (TBC) denotes a call of a block cipher (TBC), and 1 multi. denotes a multiplication on GF$(2^{n/2})$. The fourth column denotes the components that need to be implemented in parallel to achieve the latency figures, to process m-block input. The last column denotes the total size of secret key and the preprocessed values to achieve corresponding latency. For simplicity, the encryption and decryption latency of (T)BC are assumed to be identical, and (T)BC has n-bit block size and n-bit key. For Flat-OCB-f, we assume BC is AES.

Scheme	Enc latency	Dec latency	Circuit size to achieve the best latency	Total size of key and preprocessed data (bits)
ΘCB [KR11]	1 TBC call	2 TBC calls	$m + 1$ TBCs	n
Flat-ΘCB (This work)	1 TBC call	1 TBC calls	$m + 1$ TBCs	n
OCB [KR11]	2 BC calls	3 BC calls	$m + 1$ BCs	n
SIT-AE [Gue16b]	1 BC call + 1 multi.	max{1 BC call, 1 multi. }	$m + 1$ BCs and $2m$ multipliers	$2n + mn$
Flat-OCB-f (This work)	1BC call + 1 AES4 call	1BC call + 1 AES4 call	$m + 1$ BCs and one AES4	$2n + 512$
Flat-OCB-m (This work)	1BC call + 1 multi.	1BC call + 1 multi.	$m + 1$ BCs and 4 multipliers	$4n$

128-bit secret keys are XORed in each AddRoundKey individually. MASK2 computes Δ by splitting nonce into two $n/2$-bit words and multiplying them (over GF$(2^{n/2})$) with four independent $n/2$-bit keys. Let TBC-f and TBC-m denote the block cipher-based TBC defined in (4) with MASK1 (for the use of four-round AES) and MASK2 (for the use of multiplication), respectively. We also write Flat-ΘCB instantiated with TBC-f and TBC-m as Flat-OCB-f and Flat-OCB-m, respectively. By writing Flat-OCB$_{K,K'}$ or Flat-OCB, we mean both of Flat-OCB-f and Flat-OCB-m, where $K' = (K_1, K_2, K_3, K_4)$.

Properties. As shown in Table 1, the latency of Flat-ΘCB to encrypt m-block input is just one TBC invocation if $m + 1$ TBC circuits are implemented in parallel. As a mode of TBC, this latency is essentially the lowest achievable, hence optimal. Moreover, this holds for both encryption and decryption. In case of ΘCB, the decryption latency of ΘCB costs two TBC calls because it generates a tag by encrypting checksum of plaintext blocks. It can be mitigated if we change the decryption procedure so that we check the match of checksum values instead of tags (by decrypting the tag), however, this is possible only for the case of n-bit tag, which limits usability.

Comparing with ΘCB in other criteria, Flat-ΘCB has the same key size, the same number of TBC calls for encryption and decryption, and has the same security bound up to the constant (see next paragraph for the security). To get a rough idea on latency values, let us assume that a AES4 call and a multiplication on GF$(2^{n/2})$ have the same latency as one block cipher call. Then, Flat-OCB has the same encryption latency as OCB, and achieves a lower decryption latency than OCB as shown in Table 1. Note that $E_K(0^n)$ used in TBC-f and TBC-m are pre-processed, thus it increases the memory by n bits (the last column of Table 1), which will be stored at on-chip area when used in our memory encryption scheme. Although the key size of Flat-OCB is larger than that of OCB, it has the same number of block cipher calls for encryption and decryption. Regarding the security, the security bound of Flat-OCB-f decreases to $O(2^{56})$ while that of OCB is $O(2^{64})$ when $n = 128$. On the other hand, Flat-OCB-m has the same security bound as that of OCB up to the constant as well as the case of Flat-ΘCB and ΘCB. In comparison to SIT-AE, Flat-OCB have the same encryption latency and lower decryption latency, however, SIT-AE needs a circuit of $2m$ multipliers in addition

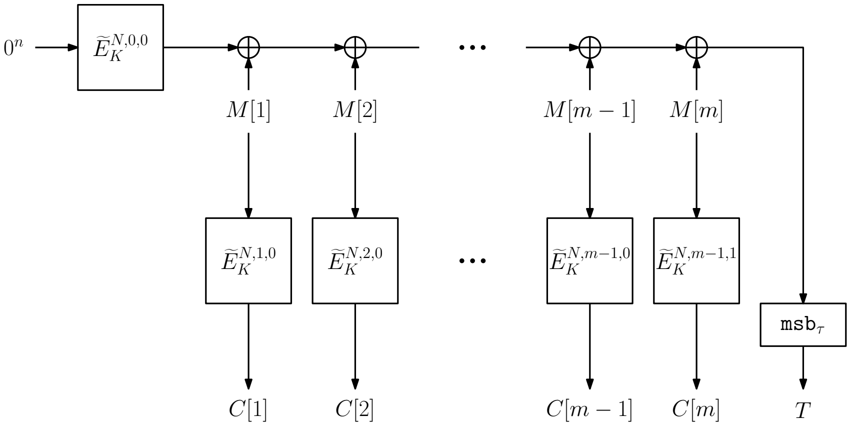

0^n
$\widetilde{E}_K^{N,0,0}$
$M[1]$
$M[2]$
$M[m-1]$
$M[m]$
$\widetilde{E}_K^{N,1,0}$
$\widetilde{E}_K^{N,2,0}$
$\widetilde{E}_K^{N,m-1,0}$
$\widetilde{E}_K^{N,m-1,1}$
$\mathtt{msb}_\tau$
$C[1]$
$C[2]$
$C[m-1]$
$C[m]$
T

Fig. 3: Flat-ΘCB, where $\widetilde{E}_K^{N,i,j}$ is a TBC with tweak (N, i, j). If we instantiate it by Alg. 8 (Alg. 9), we obtain Flat-OCB-f (Flat-OCB-m).

to $m + 1$ block cipher cores, while Flat-ΘCB requires only 4 multiplication circuits[11]. Another disadvantage of SIT-AE is its key size: it is linear to m (which will have a non-negligible impact on the overhead of the on-chip area) while that of Flat-OCB is constant.

One limitation of Flat-ΘCB and Flat-OCB is that they explicitly need integral input blocks, *i.e.*, the last block must be of n bits. ΘCB and OCB can process arbitrary length of input. By introducing a padding with a minor modification on the tweak values, Flat-ΘCB and Flat-OCB can also process arbitrary length of input. However, the length of ciphertext will expand. Anyway, this limitation is not critical for our application, where the input to AE is typically full blocks and the length is fixed.

Security. We show the security bounds of Flat-ΘCB, Flat-OCB-f and Flat-OCB-m in Theorem 2 below. As well as the case of Section 3.1, we assume that the underlying block cipher is an n-bit URP P, and only present an information-theoretic bound based on P.

In a nutshell, Flat-ΘCB has the same advantages as those of ΘCB ($\mathbf{Adv}^{\mathtt{priv}}_{\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) = 0$, $\mathbf{Adv}^{\mathtt{auth}}_{\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}^{\pm}) \le (2^{n-\tau} q_d)/(2^n - 1)$), hence there is no security penalty, up to the constant. The same applies to the advantages of Flat-OCB-m when compared with those of OCB. When $n = 128$, Flat-OCB-f has roughly 56-bit security while OCB has 64-bit security. This degradation comes from the use of the differential property of AES4 (see the proof below for the details).

We stress that the provable security of Flat-OCB-f relies solely on the pseudorandomness of AES, and AES4 does not introduce any computational assumption. This is because we used a proved AES4's (expected) differential property [KS07]; that is, it works as one large S-box with differential probability of $1/2^{113}$). The technique has been introduced by Minematsu and Tsunoo [MT06] for MAC modes, and Minematsu [Min07] for building an AES-based TBC.

Theorem 2. *The advantages of* Flat-ΘCB *and* Flat-OCB*s are*

$$\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat}\text{-}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) = 0, \qquad \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat}\text{-}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}^{\pm}) \le \frac{2q_d}{2^\tau},$$

$$\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat}\text{-}\mathsf{OCB}\text{-}\mathsf{f}_{\mathsf{P}}}(\mathcal{A}) \le \frac{2\sigma^2_{\mathtt{priv}}}{2^{113}} + \frac{2.5\sigma^2_{\mathtt{priv}}}{2^n}, \qquad \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat}\text{-}\mathsf{OCB}\text{-}\mathsf{f}_{\mathsf{P}}}(\mathcal{A}^{\pm}) \le \frac{2q_d}{2^\tau} + \frac{2\sigma^2_{\mathtt{auth}}}{2^{113}} + \frac{2.5\sigma^2_{\mathtt{auth}}}{2^n},$$

$$\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat}\text{-}\mathsf{OCB}\text{-}\mathsf{m}_{\mathsf{P}}}(\mathcal{A}) \le \frac{4.5\sigma^2_{\mathtt{priv}}}{2^n}, \qquad \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat}\text{-}\mathsf{OCB}\text{-}\mathsf{m}_{\mathsf{P}}}(\mathcal{A}^{\pm}) \le \frac{2q_d}{2^\tau} + \frac{4.5\sigma^2_{\mathtt{auth}}}{2^n},$$

[11] The number of multipliers for hardware implementation is determined depending on the system constraint/architecture in practice. We discuss its details in Section 6.

where $\mathcal{A}$ (resp. $\mathcal{A}^{\pm}$) is the adversary performing the privacy (resp. authenticity) game, and $\sigma_{\mathtt{priv}}$, $\sigma_{\mathtt{auth}}$, and q_d are the parameters for $\mathcal{A}$ and $\mathcal{A}^{\pm}$. The parameter $\sigma_{\mathtt{priv}}$ (resp. $\sigma_{\mathtt{auth}}$) is the number of the access to P *in the privacy (resp. authenticity) game such that $\sigma_{\mathtt{priv}}, \sigma_{\mathtt{auth}} \leq 2^{n-1}$. The parameter q_d is the number of the queries to the decryption oracle in the authenticity game.*

Proof. First, we evaluate the security bounds of $\mathsf{Flat\text{-}}\Theta\mathsf{CB}$, then we derive the security bounds of (two versions of) $\mathsf{Flat\text{-}OCB}$ by evaluating the security bounds of $\mathsf{TBC\text{-}f}$ and $\mathsf{TBC\text{-}m}$. Suppose that all plaintexts M and ciphertexts C in the following proof satisfies $|M| \pmod{n} = 0$ and $|C| \pmod{n} = 0$.

Proof of $\mathsf{Flat\text{-}}\Theta\mathsf{CB}$. We first evaluate the privacy bound. Since the adversary is nonce-respecting, every TURP calls in the privacy game takes different tweaks. Thus, we obtain $\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat\text{-}}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) = 0$.

We then evaluate the authenticity bound. We start with the case $q_d = 1$. Suppose that the adversary performs a decryption query after all encryption queries without loss of generality. Let $Z = \{(N_1, M_1, C_1, T_1), \ldots, (N_{q_e}, M_{q_e}, C_{q_e}, T_{q_e})\}$ be the transcript obtained by encryption queries. Let (N', C', T') be the decryption query. Suppose that T^* and M^* be the valid tag and plaintext corresponding to (N', C'), respectively. Seeing Z as a random variable, we obtain $\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat\text{-}}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}^{\pm}) = \sum_z \Pr[T' = T^* \mid Z = z] \Pr[Z = z]$. In what follows, we evaluate $\Pr[T' = T^* \mid Z = z]$ for the following cases.

1. Let $\forall i \in \{1, \ldots, q_e\}, N' \neq N_i$.
   The TURP which encrypts nonce takes a different tweak from all the tweaks invoked in the encryption queries. Thus, we obtain $\Pr[T' = T^* \mid Z = z] \leq 1/2^{\tau}$.
2. Let $\exists \alpha \in \{1, \ldots, q_e\}, N' = N_\alpha$ and $|C'|_n \neq |C_\alpha|_n$.
   We define $|C'|_n = m'$. Since the inverse of TURP which decrypts $C'[m']$ takes a different tweak from all tweaks invoked in the encryption queries, we obtain $\Pr[T' = T^* \mid Z = z] \leq 1/2^{\tau}$.
3. Let $\exists \alpha \in \{1, \ldots, q_e\}, N' = N_\alpha$ and $|C'|_n = |C_\alpha|_n$.
   We define $|C'|_n = |C_\alpha|_n = m'$ again.
   (a) When $C'[i] \neq C_\alpha[i]$ for $\exists! i \in \{1, \ldots, m'\}$ and $C'[j] = C_\alpha[j]$ for $\forall j \in \{1, \ldots, m'\} \setminus \{i\}$, it necessarily holds that $M^*[i] \neq M_\alpha[i]$ and $M^*[j] = M_\alpha[j]$. Thus, we obtain $\Pr[T' = T^* \mid Z = z] = \Pr[T' = T_\alpha \oplus \mathtt{msb}_\tau(M_\alpha[i] \oplus M^*[i]) \mid Z = z] \leq 2/2^{\tau}$.

   (b) When $C'[i] \neq C_\alpha[i]$ and $C'[j] \neq C_\alpha[j]$ for $i, j \in \{1, \ldots, m'\}$, it holds that
   $$T^* = T_\alpha \oplus \mathtt{msb}_\tau(M_\alpha[i] \oplus M^*[i]) \oplus \mathtt{msb}_\tau(M_\alpha[j] \oplus M^*[j]) \oplus \delta,$$
   where $\delta = \bigoplus_{k \neq i,j}^{m'} \mathtt{msb}_\tau(M^*[k] \oplus M_\alpha[k])$. Thus, we obtain $\Pr[T' = T^* \mid Z = z] \leq 2/2^{\tau}$.

From the above cases, we obtain the following advantage for the case $q_d = 1$:

$$\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat\text{-}}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}^{\pm}) \leq \sum_z \max_z \Pr[T' = T^* \mid Z = z] \Pr[Z = z] \leq \frac{2}{2^{\tau}}.$$

Finally, we apply the standard conversion from single to multiple decryption queries [BDJR97] and obtain the bound $q_d\,(2/2^{\tau})$ for $q_d \geq 1$. This concludes the proof for $\mathsf{Flat\text{-}}\Theta\mathsf{CB}$.

Proof of $\mathsf{Flat\text{-}OCB}$. Due to the definition of $\mathsf{Flat\text{-}OCB}$, we obtain the following inequations.

$$\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat\text{-}OCB}_{\mathsf{P}}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat\text{-}}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{tprp}}_{\mathsf{TBC}_{\mathsf{P}}}(\mathcal{B}),$$
$$\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat\text{-}OCB}_{\mathsf{P}}}(\mathcal{A}^{\pm}) \leq \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat\text{-}}\Theta\mathsf{CB}_{\widetilde{\mathsf{P}}}}(\mathcal{A}^{\pm}) + \mathbf{Adv}^{\mathrm{tsprp}}_{\mathsf{TBC}_{\mathsf{P}}}(\mathcal{B}^{\pm}),$$

where TBC is $\mathsf{TBC\text{-}f}$ when $\mathsf{Flat\text{-}OCB}$ indicates $\mathsf{Flat\text{-}OCB\text{-}f}$, and TBC is $\mathsf{TBC\text{-}m}$ when $\mathsf{Flat\text{-}OCB}$ indicates $\mathsf{Flat\text{-}OCB\text{-}m}$. Also, $\mathcal{B}$ (resp. $\mathcal{B}^{\pm}$) is the adversary against TBC querying the encryption oracle (resp. the encryption and decryption oracles). Since we have evaluated $\mathbf{Adv}^{\mathtt{priv}}_{\mathsf{Flat\text{-}OCB}_{\mathsf{P}}}(\mathcal{A})$ and $\mathbf{Adv}^{\mathtt{auth}}_{\mathsf{Flat\text{-}OCB}_{\mathsf{P}}}(\mathcal{A})$ in the previous paragraph, all that remains is to evaluate the security bounds of TBC. As well as the case of MAC, we use the methodology proposed in [MM09]. We define the offset function F of TBC as follows.

$$F_{K''}((N,i,j),\mathsf{P}(0^n)) = \mathsf{MASK}_{K''}(N) \oplus 2^i \cdot 3^j \mathsf{P}(0^n),$$

where $i \in \{0,1,2,\ldots\}$, $j \in \{0,1\}$, and $\mathsf{MASK}_{K''}(N)$ is defined as Alg. 8 (resp. Alg. 9) when $\mathsf{TBC} = \mathsf{TBC\text{-}f}$ (resp. $\mathsf{TBC} = \mathsf{TBC\text{-}m}$). Thus, we can redefine TBC using above F as $\mathsf{TBC}_{\mathsf{P}}^{N,i,j}(M) = \mathsf{P}(M \oplus F_{K''}((N,i,j),\mathsf{P}(0^n))) \oplus F_{K''}((N,i,j),\mathsf{P}(0^n))$. We again utilize the following lemma derived from Theorem 4.1 in [MM09].

Lemma 2 (A Simplified Version of Theorem 4.1 in [MM09]). *Suppose that* TBC *uses an* $(\varepsilon,\gamma,\rho)$*-uniform offset function* F*. We obtain following evaluation.*

$$\mathbf{Adv}_{\mathsf{TBC}}^{\mathrm{tsprp}}(\mathcal{B}^{\pm}) \leq q^2\left(2\varepsilon + \gamma + \rho + \frac{1}{2^{n+1}}\right),$$

where q *is the number of encryption/decryption queries such that* $q \leq 2^{n-1}$*.*

Since $\mathbf{Adv}_{\mathsf{TBC}}^{\mathrm{tprp}}(\mathcal{B}) \leq \mathbf{Adv}_{\mathsf{TBC}}^{\mathrm{tsprp}}(\mathcal{B}^{\pm})$ always holds, we derive the security bounds of $\mathsf{TBC\text{-}f}$ and $\mathsf{TBC\text{-}m}$ by evaluating the tuple $(\varepsilon,\gamma,\rho)$ in Definition 1. We first evaluate the uniformity and XOR universality of $\mathsf{MASK1}$ and $\mathsf{MASK2}$. For $\mathsf{MASK1}$, we obtain $\max_{\delta\in\{0,1\}^n} \Pr[\mathsf{MASK1}_{K''}(N) = \delta] \leq 1/2^n$ since $K_1,K_2,K_3,K_4 \in \{0,1\}^n$ are uniformly random and independent from each other. Moreover, $\max_{N\neq N',\delta\in\{0,1\}^n} \Pr[\mathsf{MASK1}_{K''}(N) \oplus \mathsf{MASK}_{K''}(N') = \delta] \leq 1/2^{113}$ holds because it is proved that the expected differential probability of $\mathsf{AES4}$ whose first-round key is 0^n is at most $1/2^{113}$ as shown by Keliher and Sui [KS07] (see also [MT06,Min07]). On the other hand, $\max_{\delta\in\{0,1\}^n} \Pr[\mathsf{MASK2}_{K''}(N) = \delta] \leq 1/2^n$ and $\max_{N\neq N',\delta\in\{0,1\}^n} \Pr[\mathsf{MASK2}_{K''}(N) \oplus \mathsf{MASK}_{K''}(N') = \delta] \leq 1/2^n$ hold due to the uniformity and independence of $K_1,K_2,K_3,K_4 \in \{0,1\}^{n/2}$. Thus, $\mathsf{MASK}_{K''}$ is γ'-uniform and ε'-almost XOR universal (AXU), where $(\gamma',\varepsilon') = (1/2^n, 1/2^{113})$ when $\mathsf{MASK} = \mathsf{MASK1}$, and $(\gamma',\varepsilon') = (1/2^n, 1/2^n)$ when $\mathsf{MASK} = \mathsf{MASK2}$.

Bound of ε. For all $\delta \in \{0,1\}^n$, we bound the probability $\Pr[X(\delta)] \coloneqq \Pr[F((N,i,j),\mathsf{P}(0^n)) \oplus F((N',i',j'),\mathsf{P}(0^n)) = \delta]$. When $N = N'$, $(i,j) \neq (i',j')$ must hold. Here, it necessarily holds that $2^i \cdot 3^j \neq 2^{i'} \cdot 3^{j'}$ due to the definition of $\mathrm{GF}(2^{128})$ described in Section 2 [Rog04]. Thus, $\Pr[X(\delta)] = \Pr[(2^i 3^j \oplus 2^{i'} 3^{j'})\mathsf{P}(0^n) = \delta] \leq 1/2^n$ holds since $\mathsf{P}(0^n)$ is uniformly random. When $N \neq N'$, we immediately obtain $\Pr[X(\delta)] \leq \varepsilon'$ since MASK is ε'-AXU. Thus, $\varepsilon = \varepsilon'$ holds.

Bound of γ. Since $\mathsf{P}(0^n)$ is uniformly random and independent from $\mathsf{MASK}_{K''}$, we obtain $\Pr[F((N,i,j),\mathsf{P}(0^n)) = \delta] = \Pr[\mathsf{MASK}_{K''}(N) \oplus 2^i \cdot 3^j \mathsf{P}(0^n) = \delta] \leq 1/2^n$.

Bound of ρ. For all $\delta \in \{0,1\}^n$, we bound the probability $\Pr[Y(\delta)] \coloneqq \Pr[F((N,i,j),\mathsf{P}(0^n)) \oplus \mathsf{P}(0^n) = \delta]$. When $(i,j) = (0,0)$, $\Pr[Y(\delta)] = \Pr[\mathsf{MASK}_{K''}(N) = \delta] \leq \gamma'$ holds due to the uniformity of MASK. When $(i,j) \neq (0,0)$, $\Pr[Y(\delta)] = \Pr[\mathsf{MASK}_{K''}(N) \oplus (2^i 3^j \oplus 1)\mathsf{P}(0^n) = \delta]$ holds. From the similar discussion of γ, we obtain $\Pr[Y(\delta)] \leq 1/2^n$. Thus, $\rho = \gamma'$ holds.

From above discussions, we obtain

$$\mathbf{Adv}_{\mathsf{TBC\text{-}f_P}}^{\mathrm{tprp}}(\mathcal{B}) \leq \mathbf{Adv}_{\mathsf{TBC\text{-}f_P}}^{\mathrm{tsprp}}(\mathcal{B}^{\pm}) \leq \frac{2\sigma_{\mathsf{ae}}^2}{2^{113}} + \frac{2.5\sigma_{\mathsf{ae}}^2}{2^n},$$
$$\mathbf{Adv}_{\mathsf{TBC\text{-}m_P}}^{\mathrm{tprp}}(\mathcal{B}) \leq \mathbf{Adv}_{\mathsf{TBC\text{-}m_P}}^{\mathrm{tsprp}}(\mathcal{B}^{\pm}) \leq \frac{4.5\sigma_{\mathsf{ae}}^2}{2^n},$$

where σ_{ae} is the number of accesses to P and $\sigma_{\mathsf{ae}} \leq 2^{n-1}$. Combining the above bounds of $\mathsf{TBC_P}$ and the bounds of $\mathsf{Flat\text{-}\Theta CB}$ proved in the previous paragraph, we obtain the security bounds of Theorem 2.

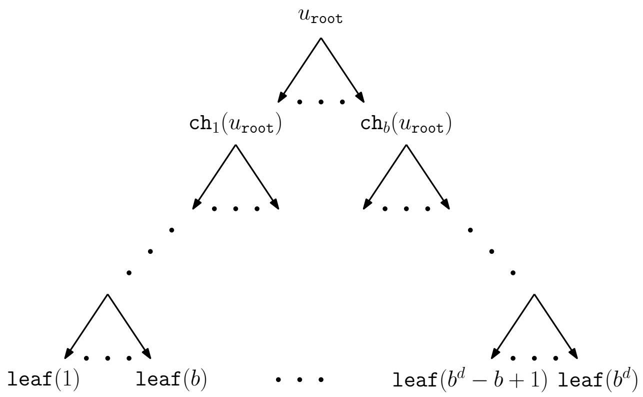

$u_{\texttt{root}}$
$\texttt{ch}_1(u_{\texttt{root}})$
$\texttt{ch}_b(u_{\texttt{root}})$
$\texttt{leaf}(1)$
$\texttt{leaf}(b)$
$\texttt{leaf}(b^d - b + 1)$
$\texttt{leaf}(b^d)$

Fig. 4: Tree structure for ELM.

4 ELM

In this section, we detail our authentication tree scheme, ELM. As described before, we employ the tree construction PAT2. The inner MAC and AE schemes are instantiated by PXOR-MAC and Flat-OCB. Let ELM1 and ELM2 be the instances of ELM employing Flat-OCB-f and Flat-OCB-m as the inner AE schemes, respectively. We show how to combine PAT2, PXOR-MAC, and Flat-OCB in an optimal manner to reduce latency and computation for updating tree.

4.1 Notations for Tree

We describe a tree structure for ELM in Fig. 4. The number of branches is denoted by $b \geq 2$, and d denotes the depth, where the depth of root and a leaf node are defined as 0 and d, respectively. We assume a balanced tree, hence the number of leaf nodes is b^d. The entire memory (plaintext) to protect is divided into chunks, where each chunk has ℓ bits. We associate a chunk with each leaf node denoted by $\mathsf{leaf}(i)$ for $1 \leq i \leq b^d$. Thus, the whole plaintext M to be protected by a authentication tree scheme consists of $M = M[1] \parallel \cdots \parallel M[b^d]$ such that $|M[i]| = \ell$ for $1 \leq i \leq b^d$ and $M[i]$ is associated with $\mathsf{leaf}(i)$. The ciphertext chunk corresponding to $M[i]$ is denoted by $C[i]$, which is stored in $\mathsf{leaf}(i)$. For the node u, the memory address, the counter, and the tag are denoted by $\mathsf{ADD}(u)$, $\mathsf{CTR}(u)$, $\mathsf{Tag}(u)$, respectively. The lengths of the memory address, the counter, and the tag are α, β, and τ, respectively. All the data stored in the on-chip and off-chip area is denoted by σ, which includes $C[i]$ for $1 \leq i \leq b^d$, and $\mathsf{CTR}(u)$ and $\mathsf{Tag}(u)$ for all nodes u. As we adopt PAT2, we exclude the node addresses from σ and assume that they are given by the system when needed. Suppose the root node is denoted by u_{root}, we store $\mathsf{CTR}(u_{\mathsf{root}})$ in the on-chip area. The leftover data of σ is stored off-chip.

We may also use σ to mean the tree construction itself. We also write a node u, leaf node, plaintext chunk, ciphertext chunk of σ as u^σ, $\mathsf{leaf}(i)^\sigma$, $M^\sigma[i]$, and $C^\sigma[i]$, respectively. If no confusion is possible, we omit their superscript σ. For any non-leaf node u^σ and $i \in \{1, \dots, b\}$, $\mathsf{ch}_i(u^\sigma)$ denotes its i-th child node.

4.2 Specification of ELM

ELM consists of three algorithms: InitTree, Verify, and Update defined in Algs. 10, 11, and 12, which is denoted by $\mathsf{ELM} = (\mathsf{InitTree}, \mathsf{Verify}, \mathsf{Update})$. Suppose that they take the tuple of keys for AE and MAC $K_T = ((K_1, K'_1), (K_2, K'_2))$ as input. The algorithm InitTree is the initialization of the tree. It takes a plaintext M and a tuple of keys as input, and outputs a tree σ. Here, σ consists of the local counters being initialized to zero, the tags for intermediate nodes, and the (ciphertext, tag) pairs for the leaf nodes. We use the incremental property of MAC (line 11) to efficiently compute the tags for the intermediate nodes since all message inputs to PXOR-MAC are identical (all zero). The algorithm Verify checks the validity of a specified leaf node. It is associated with a memory read operation. Verify takes the index of a leaf node

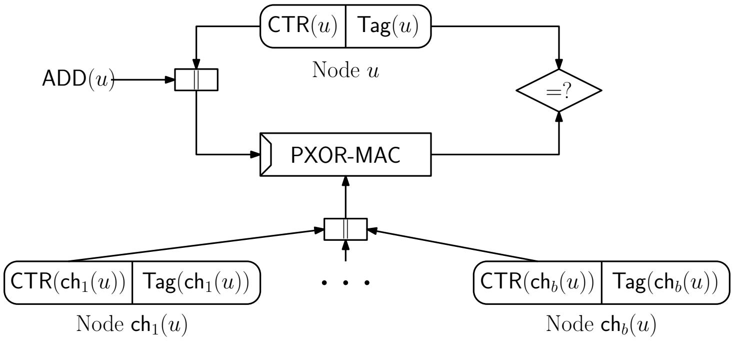


Fig. 5: A part of a verification procedure at an internal node u.

idx $(1 \leq idx \leq b^d)$ and the tree σ as input. The algorithm returns $\top$ if all the verifications of PXOR-MAC and the decryption of Flat-OCB in Alg. 11 are successful, and otherwise returns $\bot$. The algorithm Update checks the validity of a specified leaf node, and if the verification is successful, updates the leaf node by re-encrypting the leaf node with a new plaintext. It also updates the tags and the counters of the nodes on the corresponding path from the leaf to the root. It is associated with a memory write operation. Note that it is essential for Update to check the validity of the data associated in node path in order to prevent a replay attack[12]. Update takes the index of leaf node idx $(1 \leq idx \leq b^d)$, the update value (new plaintext) B such that $|B| = \ell$, and the tree σ as input. It returns a renewed tree $\tilde{\sigma}$ if the verification is successful, otherwise $\bot$. For the verification and update of intermediate nodes in Update, we use PXOR-MAC.$\mathcal{VU}$ defined in Alg. 13. It combines PXOR-MAC.$\mathcal{V}$ and PXOR-MAC.$\mathcal{U}$ and prunes some redundant block cipher calls that would be imposed if we invoked PXOR-MAC.$\mathcal{V}$ and PXOR-MAC.$\mathcal{U}$ in a black-box way. In a similar manner to PXOR-MAC.$\mathcal{U}$ in Section 3.1, the input tuple of PXOR-MAC.$\mathcal{VU}$ can also be described as the set of block indices to replace and the contents of new blocks, in addition to $(N_{\mathrm{old}}, M_{\mathrm{old}}, T_{\mathrm{old}})$.

When we use Flat-OCB-m in lines 13–16 of Alg. 12, there are some redundant field multiplications for deriving $\varDelta$ if $\mathtt{msb}_{n/2}(\mathsf{ADD}(u_d^{\sigma}) \,\|\, \mathsf{CTR}(u_d^{\sigma})) = \mathtt{msb}_{n/2}(\mathsf{ADD}(u_d^{\sigma}) \,\|\, \mathsf{CTR}(u_d^{\tilde{\sigma}}))$ or $\mathtt{lsb}_{n/2}(\mathsf{ADD}(u_d^{\sigma}) \,\|\, \mathsf{CTR}(u_d^{\sigma})) = \mathtt{lsb}_{n/2}(\mathsf{ADD}(u_d^{\sigma}) \,\|\, \mathsf{CTR}(u_d^{\tilde{\sigma}}))$ holds. These can be saved by caching in the same manner to the case of PXOR-MAC.$\mathcal{VU}$.

4.3 Features

ELM is designed to achieve low latency by utilizing the incremental property of MAC and full parallelizability of the cryptographic components and the tree structure. Especially, incremental property greatly contributes to reduced latency of Update. Since Update includes the operation of Verify and plain update of nodes, we can use the incremental MAC of *basic* security as described in Section 3.1. Moreover, rather than naively applying an incremental MAC, we optimize Update by defining PXOR-MAC.$\mathcal{VU}$ so that we can save some redundant computations generated by the invocations of both verification and update, which will contribute to reducing latency. Suppose $\alpha = \beta = n/2$ and some even b. One invocation of Verify needs $(1 + 2/b)d$ block cipher calls for intermediate and root nodes. One invocation of Update needs $(3 + 2/b)d$ block cipher calls for intermediate and root nodes, while Update with a non-incremental MAC needs at least twice as many block cipher calls as Verify does. In addition, ELM is also scalable in terms of on-chip size. It is because the sizes of key and preprocessed data are constant. Suppose that the key of block cipher is n bits, ELM1 and ELM2 need $5n + 512$ bits and $7n$ bits for key and preprocessed data, respectively. Thus, the required on-chip memory is small for any parameter of the tree. However, SIT (here we mean a generalized version, *i.e.*, PAT2 with the MAC and AE schemes used by SIT) needs on-chip area of size linear to b and β.

[12] One of the reasons why the adversary cannot mount a replay attack against PAT2 is CTR($\cdot$) has the one-time property (see Section 5.1 for the details). If the verification in Update is bypassed, the adversary can roll back the value of CTR($\cdot$) and mount a replay attack.

Up to this point, we have ignored the off-chip memory overhead caused by storing counters and tags. However, it may be non-negligible if the target memory size gets larger. In such a case, we can combine ELM and a well-known technique to reduce the memory needed for counters, called *Split counter* [YEP+06]. The technique will incur an increased average latency and has been adopted by state-of-the-art schemes [TSB18, SNR+18]. Fortunately, the incremental property of PXOR-MAC is still quite effective even if we adopt the split counter. See Section 7.2 for more details.

Algorithm 10 InitTree: Initialization of the tree construction σ

```
Input  $K_T = ((K_1, K'_1), (K_2, K'_2)),\ M = M[1] \,\|\, \cdots \,\|\, M[b^d]$ s.t$|M[i]| = \ell$ for $1 \le i \le b^d$
Output  $\sigma$
$\sigma \leftarrow 0^{\left(\frac{b^d-1}{b-1}\right)\times(\beta+\tau)+b^d\times(\ell+\beta+\tau))}$
for all nodes $u$ do
    $\mathsf{CTR}(u^\sigma) \leftarrow 0^{\beta-1}1$
end for
for $1 \le i \le b^d$ do
    $(C[i], \mathsf{Tag}(\mathsf{leaf}(i)^\sigma)) \leftarrow \mathsf{Flat\text{-}OCB}.\mathcal{E}_{K_1,K'_1}(\mathsf{ADD}(\mathsf{leaf}(i)^\sigma) \,\|\, \mathsf{CTR}(\mathsf{leaf}(i)^\sigma), M[i])$
end for
$N_{\mathrm{old}} \leftarrow \mathsf{ADD}(u^\sigma_{\mathsf{root}}) \,\|\, \mathsf{CTR}(u^\sigma_{\mathsf{root}}),\ M_{\mathrm{old}} \leftarrow \mathsf{CTR}(\mathsf{ch}_1(u^\sigma_{\mathsf{root}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u^\sigma_{\mathsf{root}}))$
$\mathsf{Tag}(u^\sigma_{\mathsf{root}}) \leftarrow \mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K_2,K'_2}(N_{\mathrm{old}}, M_{\mathrm{old}})$
for all intermediate nodes $u$ do
    $\mathsf{Tag}(u^\sigma) \leftarrow \mathsf{PXOR\text{-}MAC}.\mathcal{U}_{K_2,K'_2}(N_{\mathrm{old}}, M_{\mathrm{old}}, \mathsf{Tag}(u^\sigma_{\mathsf{root}}), \mathsf{ADD}(u^\sigma) \,\|\, \mathsf{CTR}(u^\sigma), M_{\mathrm{old}})$
end for
return  $\sigma$
```

Algorithm 11 Verify : Checking the validity of leaf(idx). ($1 \le idx \le b^d$)

```
Input  $K_T = ((K_1, K'_1), (K_2, K'_2)),\ idx,\ \sigma$
Output  $\top$ or $\bot$
$(u^\sigma_0, \cdots, u^\sigma_d) \leftarrow$ path of nodes from root to specified leaf
( i.e., $u^\sigma_0$ is the root node $u^\sigma_{\mathsf{root}}$, and $u^\sigma_d$ is equal to leaf($idx$). )
for $0 \le i \le d-1$ do
    if $\mathsf{PXOR\text{-}MAC}.\mathcal{V}_{K_2,K'_2}(\mathsf{ADD}(u^\sigma_i) \,\|\, \mathsf{CTR}(u^\sigma_i), \mathsf{CTR}(\mathsf{ch}_1(u^\sigma_i)) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u^\sigma_i)), \mathsf{Tag}(u^\sigma_i)) = \bot$ then
        return  $\bot$
    end if
end for
if $\mathsf{Flat\text{-}OCB}.\mathcal{D}_{K_1,K'_1}(\mathsf{ADD}(u^\sigma_d) \,\|\, \mathsf{CTR}(u^\sigma_d), C[idx], \mathsf{Tag}(u^\sigma_d)) = \bot$ then
    return  $\bot$
end if
return  $\top$
```

Algorithm 13 $\mathsf{PXOR\text{-}MAC}.\mathcal{VU}_{K,K'}(N_{\text{old}},\, M_{\text{old}},\, T_{\text{old}},\, N_{\text{new}},\, M_{\text{new}})$

```
$L \leftarrow E_K(0^n)$, $T' \leftarrow 0^\tau$, $T_{\text{new}} \leftarrow T_{\text{old}}$, $\Sigma \leftarrow 0^\tau$, $\mathbb{E} \leftarrow \varepsilon$
$M_{\text{old}}[1] \| \cdots \| M_{\text{old}}[m] \xleftarrow{n} M_{\text{old}}$, $M_{\text{new}}[1] \| \cdots \| M_{\text{new}}[m] \xleftarrow{n} M_{\text{new}}$
for $1 \le i \le m$ do
  $\Sigma' \leftarrow \mathtt{msb}_\tau(E_K(M_{\text{old}}[i] \oplus K' \cdot i))$
  $T' \leftarrow T' \oplus \Sigma'$
  if $M_{\text{old}}[i] \neq M_{\text{new}}[i]$ then
    $\mathbb{E} \xleftarrow{\cup} \{i\}$, $\Sigma \leftarrow \Sigma \oplus \Sigma'$
  end if
end for
$T' \leftarrow T' \oplus \mathtt{msb}_\tau(E_K(N_{\text{old}} \oplus K' \cdot m \oplus L))$
if $N_{\text{old}} \neq N_{\text{new}}$ then
  $\mathbb{E} \xleftarrow{\cup} \{m+1\}$, $\Sigma \leftarrow \Sigma \oplus \mathtt{msb}_\tau(E_K(N_{\text{old}} \oplus K' \cdot m \oplus L))$
end if
if $T' \neq T_{\text{old}}$ then
  return $\perp$
end if
$T_{\text{new}} \leftarrow T_{\text{new}} \oplus \Sigma$
for $i \in \mathbb{E}$ do
  $T_{\text{new}} \leftarrow T_{\text{new}} \oplus \mathtt{msb}_\tau(E_K(M_{\text{new}}[i] \oplus K' \cdot i))$
end for
if $m+1 \in \mathbb{E}$ then
  $T_{\text{new}} \leftarrow T_{\text{new}} \oplus \mathtt{msb}_\tau(E_K(N_{\text{new}} \oplus K' \cdot m \oplus L))$
end if
return $T_{\text{new}}$
```

Algorithm 12 Update : Update the message of $\mathsf{leaf}(idx)$ to B. ($1 \le idx \le b^d$)

```
Input $K_T = ((K_1, K'_1), (K_2, K'_2))$, $idx$, $B$, $\sigma$
Output $\tilde{\sigma}$ or $\perp$
$\tilde{\sigma} \leftarrow \sigma$, $(u_0, \cdots, u_d) \leftarrow$ path of nodes from root to specified leaf
for $0 \le i \le d$ do
  $\mathsf{CTR}(u_i^{\tilde{\sigma}}) \leftarrow \mathsf{CTR}(u_i^{\tilde{\sigma}}) + 1$
end for
for $0 \le i \le d-1$ do
  $N_{\text{old}} \leftarrow \mathsf{ADD}(u_i^\sigma) \| \mathsf{CTR}(u_i^\sigma)$, $M_{\text{old}} \leftarrow \mathsf{CTR}(\mathsf{ch}_1(u_i^\sigma)) \| \cdots \| \mathsf{CTR}(\mathsf{ch}_b(u_i^\sigma))$
  $N_{\text{new}} \leftarrow \mathsf{ADD}(u_i^\sigma) \| \mathsf{CTR}(u_i^{\tilde{\sigma}})$, $M_{\text{new}} \leftarrow \mathsf{CTR}(\mathsf{ch}_1(u_i^{\tilde{\sigma}})) \| \cdots \| \mathsf{CTR}(\mathsf{ch}_b(u_i^{\tilde{\sigma}}))$
  $\mathsf{Tag}(u_i^{\tilde{\sigma}}) \leftarrow \mathsf{PXOR\text{-}MAC}.\mathcal{VU}_{K_2,K'_2}(N_{\text{old}}, M_{\text{old}}, \mathsf{Tag}(u_i^\sigma), N_{\text{new}}, M_{\text{new}})$
  if $\mathsf{Tag}(u_i^{\tilde{\sigma}}) = \perp$ then
    return $\perp$
  end if
end for
if $\mathsf{Flat\text{-}OCB}.\mathcal{D}_{K_1,K'_1}(\mathsf{ADD}(u_d^\sigma) \| \mathsf{CTR}(u_d^\sigma), C^\sigma[idx], \mathsf{Tag}(u_d^\sigma)) = \perp$ then
  return $\perp$
end if
$(C^{\tilde{\sigma}}[idx], \mathsf{Tag}(u_d^{\tilde{\sigma}})) \leftarrow \mathsf{Flat\text{-}OCB}.\mathcal{E}_{K_1,K'_1}(\mathsf{ADD}(u_d^\sigma) \| \mathsf{CTR}(u_d^{\tilde{\sigma}}), B)$
return $\tilde{\sigma}$
```

5 Security of PAT2

In this section, we show that the security of $\mathsf{PAT2}$ can be reduced to the security of underlying MAC and AE schemes. This immediately implies the provable security of ELM. First, we define security notions of an authentication tree in Section 5.1. The privacy notion is defined analogously to that defined for nonce-based AE (Section 2), and the unforgeability notion is mostly identical to that defined in [HJ06]. Then, we evaluate the security of $\mathsf{PAT2}$ in Section 5.2.

5.1 Security Notion of Authentication Tree

Suppose that Tree is an authentication tree scheme defined as a tuple of three functions: the initialization function $\mathsf{InitTree}$, the verification function Verify, and the update function Update, denoted by $\mathsf{Tree} = (\mathsf{InitTree}, \mathsf{Verify}, \mathsf{Update})$. Recall that $\mathsf{InitTree}(M) = \sigma$, $\mathsf{Verify}(idx, \sigma) = \top$ or $\bot$, and $\mathsf{Update}(idx, B, \sigma) = \tilde{\sigma}$ or $\bot$ (See Section 4 for details). Also recall that σ includes data stored in the on-chip memory (*i.e.*, tamper-free area), which we denote $\mathsf{Sec}(\sigma)$[13].

Security notions. We define two security notions of an authentication tree: privacy and unforgeability. For the privacy of Tree, we define $\mathsf{InitTree\text{-}\$}$ and $\mathsf{Update\text{-}\$}$. They return their ciphertexts and tags to be stored in the leaf nodes as random strings whose lengths are the same as those of $\mathsf{InitTree}$ and Update, respectively. Regarding other variables, for example, the data associated with the intermediate nodes, they return the same outputs as $\mathsf{InitTree}$ and Update. The privacy security of Tree is defined as the probability that an adversary $\mathcal{A}$ successfully distinguishes $(\mathsf{InitTree}, \mathsf{Update})$ from $(\mathsf{InitTree\text{-}\$}, \mathsf{Update\text{-}\$})$. It is written as

$$\mathbf{Adv}^{\mathtt{ptree}}_{\mathsf{Tree}}(\mathcal{A}) := |\Pr[\mathcal{A}^{\mathsf{InitTree},\mathsf{Update}} \to 1] - \Pr[\mathcal{A}^{\mathsf{InitTree\text{-}\$},\mathsf{Update\text{-}\$}} \to 1]|,$$

where $\mathcal{A}$ plays the following game.

1. $\mathcal{A}$ queries M to the tree initialization oracle ($\mathsf{InitTree}$ or $\mathsf{InitTree\text{-}\$}$) and obtains $\sigma_{\mathtt{init}}$.
2. $\mathcal{A}$ makes q adaptive queries to the update oracle (Update or $\mathsf{Update\text{-}\$}$). Let $\{(idx_1, B_1, \sigma_1, \tilde{\sigma}_1), \ldots, (idx_q, B_q, \sigma_q, \tilde{\sigma}_q)\}$ be the transcript obtained by the update queries. Here, we assume that $\sigma_1 = \sigma_{\mathtt{init}}$ and $\sigma_i = \widetilde{\sigma}_{i-1}$ for $2 \le i \le q$ so that $\mathcal{A}$ can always obtain an updated tree, not $\bot$.
3. $\mathcal{A}$ guesses which the oracle pair she has queried (($\mathsf{InitTree}, \mathsf{Update}$) or ($\mathsf{InitTree\text{-}\$}, \mathsf{Update\text{-}\$}$)) and accordingly outputs a bit.

For the unforgeability notion for Tree, our definition follows [HJ06]. It is defined as the advantage of an adversary $\mathcal{A}'$ querying $\mathsf{InitTree}$ and Update successfully distinguishes Verify from $\bot_{\mathsf{Tree}}(\cdot,\cdot)$ which always returns $\bot$ for any inputs. The unforgeability advantage of $\mathcal{A}'$ is defined as

$$\mathbf{Adv}^{\mathtt{uftree}}_{\mathsf{Tree}}(\mathcal{A}') := |\Pr[\mathcal{A}'^{\mathsf{InitTree},\mathsf{Update},\mathsf{Verify}} \to 1] - \Pr[\mathcal{A}'^{\mathsf{InitTree},\mathsf{Update},\bot_{\mathsf{Tree}}} \to 1]|,$$

where $\mathcal{A}'$ plays the following game.

1. $\mathcal{A}'$ queries M to $\mathsf{InitTree}$ and obtains $\sigma_{\mathtt{init}}$.
2. $\mathcal{A}'$ makes q' adaptive queries to Update. Let $\{(idx_1, B_1, \sigma_1, \tilde{\sigma}_1), \ldots, (idx_{q'}, B_{q'}, \sigma_{q'}, \tilde{\sigma}_{q'})\}$ be the transcript obtained by update queries. As well as the privacy game, we assume that $\sigma_1 = \sigma_{\mathtt{init}}$ and $\sigma_i = \widetilde{\sigma}_{i-1}$ for $2 \le i \le q'$ so that $\mathcal{A}'$ can always obtain an updated tree, not $\bot$.
3. $\mathcal{A}'$ queries (idx', σ') to the verification oracle (Verify or $\bot_{\mathsf{Tree}}$) and obtains $\top$ or $\bot$. Let $(u_0, \ldots, u_d)$ be the path of nodes from the root node to $\mathsf{leaf}(idx')$. To exclude a trivial win, we assume that there exists $i \in \{0, \ldots, d\}$ such that $u_i^{\sigma'}$ stores different data from that stored in $u_i^{\tilde{\sigma}_{q'}}$. Moreover, $\mathsf{Sec}(\sigma') = \mathsf{Sec}(\tilde{\sigma}_{q'})$ also must hold since the data in the on-chip area cannot be tampered.

[13] In this paper, we do not assume confidentiality of $\mathsf{Sec}(\sigma)$, thus the adversary can look into it. It is a weaker assumption than that assuming both confidentiality and tamper freeness.

4. $\mathcal{A}'$ guesses which oracle pair she has queried (($\mathsf{InitTree}, \mathsf{Update}, \mathsf{Verify}$) or ($\mathsf{InitTree}$, $\mathsf{Update}, \bot_{\mathsf{Tree}}$)) and accordingly outputs a bit.

Algorithm 14 InitTree

```
Input  $K_T = (K_A, K_M)$, $M = M[1] \,\|\, \cdots \,\|\, M[b^d]$
Output  $\sigma$
$\sigma \leftarrow 0^{\left(\frac{b^d-1}{b-1}\right)\times(\beta+\tau)+b^d\times(\ell+\beta+\tau))}$
for all node $u$ do
    $\mathsf{CTR}(u^\sigma) \leftarrow 0^{\beta-1}1$
end for
for $1 \leq i \leq b^d$ do
    $(C[i], \mathsf{Tag}(\mathsf{leaf}(i)^\sigma)) \leftarrow \mathsf{AE}.\mathcal{E}_{K_A}(\mathsf{ADD}(\mathsf{leaf}(i)^\sigma) \,\|\, \mathsf{CTR}(\mathsf{leaf}(i)^\sigma), M[i])$
end for
for all non-leaf node $u$ do
    $\mathsf{Tag}(u^\sigma) \leftarrow \mathsf{MAC}.\mathcal{T}_{K_M}(\mathsf{ADD}(u^\sigma) \,\|\, \mathsf{CTR}(u^\sigma), \mathsf{CTR}(\mathsf{ch}_1(u^\sigma)) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u^\sigma)))$
end for
return $\sigma$
```

Algorithm 15 Verify

```
Input  $K_T = (K_A, K_M)$, $idx$, $\sigma$
Output  $\top$ or $\bot$
$(u_0^\sigma, \cdots, u_d^\sigma) \leftarrow$ path of nodes from root to specified leaf
    ( i.e., $u_0^\sigma$ is the root node and $u_d^\sigma$ is equal to $\mathsf{leaf}(idx)$. )
for $0 \leq i \leq d-1$ do
    if $\mathsf{MAC}.\mathcal{V}_{K_M}(\mathsf{ADD}(u_i^\sigma) \,\|\, \mathsf{CTR}(u_i^\sigma), \mathsf{CTR}(\mathsf{ch}_1(u_i^\sigma)) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^\sigma)), \mathsf{Tag}(u_i^\sigma)) = \bot$ then
        return $\bot$
    end if
end for
if $\mathsf{AE}.\mathcal{D}_{K_A}(\mathsf{ADD}(u_d^\sigma) \,\|\, \mathsf{CTR}(u_d^\sigma), C[idx], \mathsf{Tag}(u_d^\sigma)) = \bot$ then
    return $\bot$
end if
return $\top$
```

Algorithm 16 Update

```
Input  $K_T = (K_A, K_M)$, $idx$, $B$, $\sigma$
Output  $\tilde{\sigma}$ or $\bot$
if $\mathsf{Verify}(idx, \sigma) = \bot$ then
    return $\bot$
end if
$\tilde{\sigma} \leftarrow \sigma$, $(u_0, \cdots, u_d) \leftarrow$ path of nodes from root to specified leaf
for $0 \leq i \leq d$ do
    $\mathsf{CTR}(u_i^{\tilde{\sigma}}) = \mathsf{CTR}(u_i^{\tilde{\sigma}}) + 1$
end for
for $0 \leq i \leq d-1$ do
    $\mathsf{Tag}(u_i^{\tilde{\sigma}}) \leftarrow \mathsf{MAC}.\mathcal{T}_{K_M}(\mathsf{ADD}(u_i^{\tilde{\sigma}}) \,\|\, \mathsf{CTR}(u_i^{\tilde{\sigma}}), \mathsf{CTR}(\mathsf{ch}_1(u_i^{\tilde{\sigma}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\tilde{\sigma}})))$
end for
$(C[idx], \mathsf{Tag}(u_d^{\tilde{\sigma}})) \leftarrow \mathsf{AE}.\mathcal{E}_{K_A}(\mathsf{ADD}(u_d^{\tilde{\sigma}}) \,\|\, \mathsf{CTR}(u_d^{\tilde{\sigma}}), B)$
return $\tilde{\sigma}$
```

Suppose that we also write as $\sigma_{\mathtt{init}} = \tilde{\sigma}_0$. We stress that $\mathcal{A}'$ can perform verification query such that $u_i^{\sigma'}$ stores same data as that stored in $u_i^{\tilde{\sigma}_j}$ for $0 \leq i \leq d$ and $0 \leq j \leq q' - 1$, unless the data stored in $u_i^{\tilde{\sigma}_j}$ is the same as that stored in $u_i^{\tilde{\sigma}_{q'}}$ as described in the third operation of the above game. This condition is essential for the unforgeability notion to capture an adversary who performs a replay attack, which is the attack to replace data with old legitimate one.

Rationale of security notions. The privacy notion is defined similarly to that defined for nonce-based AE. Namely, we evaluate per-node indistinguishability between ciphertexts and tags associated with leaf nodes and random strings against an adversary performing chosen-plaintext attack (IND-CPA) [BN00].

For the unforgeability notion, we follow that defined in [HJ06], thus just extend it from for the authentication tree without encryption of leaf nodes to for that with. The notion captures the adversary performs CPA (by initialization query and update queries) and tampering the data stored in the off-chip area once (by verification query)[14]. This means that the unforgeability game can simulate, say, an adversary who tampers the data stored in a certain node with new values, swaps the data associated with two nodes, and performs replay attack (as described in the definition of the unforgeability game), in addition to the adversary captured by the privacy notion. Especially, it is important to capture the adversary performing replay attack since the security notion of a general MAC does not capture her. By proving the unforgeability advantage is negligible, we can prove that the authentication tree scheme can detect tampering (including replay) by such an adversary with sufficiently high probability.

5.2 Security Bounds of PAT2

Let MAC_{K_M} and AE_{K_A} be a MAC scheme and an AE scheme, where K_M and K_A are uniformly random and independent. We describe three functions of $\mathsf{PAT2}$, $(\mathsf{InitTree}, \mathsf{Verify}, \mathsf{Update})$, using MAC_{K_M} and AE_{K_A} in Algs. 14, 15, and 16, respectively. We note that, when $(\mathsf{MAC}, \mathsf{AE})$ is instantiated as $(\mathsf{PXOR\text{-}MAC}, \mathsf{Flat\text{-}OCB})$, each function of $\mathsf{PAT2}$ returns the same computation result as a corresponding function of ELM.

Privacy Bound.

Theorem 3. *The privacy advantage of* $\mathsf{PAT2}$ *is bounded as follows.*

$$\mathbf{Adv}^{\mathtt{ptree}}_{\mathsf{PAT2}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathtt{priv}}_{\mathsf{AE}}(\mathcal{A}_{\mathsf{ae}}),$$

where $\mathcal{A}_{\mathsf{ae}}$ *is a privacy adversary against* AE*, using* $b^d + q$ *queries.*

Proof. We assume that $\mathcal{A}$ is given the MAC key K_M, denoted by $\mathcal{A}(K_M)$. Since $\mathcal{A}(K_M)$ can compute data associated with the root node and intermediate nodes, we can assume that $\mathcal{A}(K_M)$ obtains only data associated with leaf nodes from the tree initialization oracle and the update oracle. Let $\mathcal{A}_{\mathsf{ae}}$ be the privacy adversary against AE. The adversary $\mathcal{A}_{\mathsf{ae}}$ can properly simulate the privacy game of $\mathcal{A}(K_M)$. In what follows, we describe how $\mathcal{A}_{\mathsf{ae}}$ simulates two oracles that $\mathcal{A}(K_M)$ queries. If $\mathcal{A}(K_M)$ queries $\mathsf{InitTree}$ (resp. $\mathsf{InitTree\text{-}\$}$), $\mathcal{A}_{\mathsf{ae}}$ can simulate it by querying $\mathsf{AE}.\mathcal{E}$ (resp. \$ defined in Section 2) in the same manner as Alg. 14. Note that the tree initialization query of $\mathcal{A}(K_M)$ invokes nonce-respecting encryption queries of $\mathcal{A}_{\mathsf{ae}}$ since $\mathsf{ADD}(\mathsf{leaf}(i)) \,\|\, \mathsf{CTR}(\mathsf{leaf}(i)) \neq \mathsf{ADD}(\mathsf{leaf}(j)) \,\|\, \mathsf{CTR}(\mathsf{leaf}(j))$ necessarily holds for $1 \leq i \neq j \leq b^d$. Thus, the privacy adversary $\mathcal{A}_{\mathsf{ae}}$ can simulate the initialization oracles for $\mathcal{A}(K_M)$. Regarding the queries to the update oracles, $\mathcal{A}(K_M)$ invokes decryption queries of AE since the update queries invoke the verification function of the authentication tree (line 1 in Alg. 16). However, the verification function always outputs $\top$ since $\sigma_1 = \sigma_{\mathtt{init}}$ and $\sigma_i = \tilde{\sigma}_{i-1}$ for $2 \leq i \leq q$ as defined in Section 5.1. Thus, $\mathcal{A}_{\mathsf{ae}}$ can always output $\top$ regardless of inputs to simulate the subroutine verification function in update queries. The adversary $\mathcal{A}_{\mathsf{ae}}$ can simulate the

[14] As well as the game defined in [HJ06], the game defined in this paper does not capture the adversary performing multiple tampering attacks (*i.e.*, multiple verification queries). To do so, we have to define a system operating behavior when a verification process fails (*e.g.*, restarting from initialization process or removing the data which is possibly tampered without invoking initialization process). However, it is depends heavily on actual systems.

leftover pure update function (line 4 – 12 in Alg. 16) in the same manner as the simulation of the initialization oracles: if $\mathcal{A}(K_M)$ queries Update (resp. $\mathsf{Update\text{-}\$}$), $\mathcal{A}_{\mathsf{ae}}$ can simulate it by querying $\mathsf{AE}.\mathcal{E}$ (resp. \$) in the same manner as Alg. 14. Also, the update query of $\mathcal{A}(K_M)$ invokes nonce-respecting encryption query of $\mathcal{A}_{\mathsf{ae}}$ due to the node-unique property of $\mathsf{ADD}(\cdot)$ and the one-time property of $\mathsf{CTR}(\cdot)$. Thus, the privacy adversary $\mathcal{A}_{\mathsf{ae}}$ can simulate the update oracles for $\mathcal{A}(K_M)$. Finally, we can confirm easily that the sequence of queries in the privacy game of $\mathcal{A}(K_M)$ invokes nonce-respecting encryption queries of $\mathcal{A}_{\mathsf{ae}}$, hence the privacy adversary against AE $\mathcal{A}_{\mathsf{ae}}$ can properly simulate $\mathcal{A}(K_M)$ and we obtain following evaluations.

$$\begin{aligned}\mathbf{Adv}^{\mathtt{ptree}}_{\mathsf{Tree}}(\mathcal{A}) &= |\Pr[\mathcal{A}^{\mathsf{InitTree},\mathsf{Update}} \to 1] - \Pr[\mathcal{A}^{\mathsf{InitTree\text{-}\$},\mathsf{Update\text{-}\$}} \to 1]| \\ &\leq |\Pr[\mathcal{A}(K_M)^{\mathsf{InitTree},\mathsf{Update}} \to 1] - \Pr[\mathcal{A}(K_M)^{\mathsf{InitTree\text{-}\$},\mathsf{Update\text{-}\$}} \to 1]| \\ &= \mathbf{Adv}^{\mathtt{priv}}_{\mathsf{AE}}(\mathcal{A}_{\mathsf{ae}}),\end{aligned}$$

where $\mathcal{A}_{\mathsf{ae}}$ queries $b^d + q$ to encryption oracle because a initialization query of $\mathcal{A}(K_M)$ invokes b^d times encryption queries of $\mathcal{A}_{\mathsf{ae}}$ and update queries of $\mathcal{A}(K_M)$ invoke q times encryption queries of $\mathcal{A}_{\mathsf{ae}}$.

Unforgeability Bound.

Theorem 4. *The unforgeability advantage of* $\mathsf{PAT2}$ *is bounded as follows.*

$$\mathbf{Adv}^{\mathtt{uftree}}_{\mathsf{Tree}}(\mathcal{A}') \leq \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{AE}}(\mathcal{A}^{\pm}_{\mathsf{ae}}) + \mathbf{Adv}^{\mathtt{mac}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{mac}}),$$

where $\mathcal{A}^{\pm}_{\mathsf{ae}}$ *is the authenticity adversary against* AE *and* $\mathcal{A}_{mac}$ *is the adversary against* MAC*. The adversary* $\mathcal{A}^{\pm}_{\mathsf{ae}}$ *queries* $b^d + q'$ *times to the encryption oracle and queries one time to the decryption oracle. The adversary* $\mathcal{A}_{mac}$ *queries* $(b^d - 1)/(b-1) + q'd$ *times to the tagging oracle and queries* d *times to the MAC verification oracle.*

Proof. Let $\mathcal{A}'_{\mathtt{ma}}$ denote an adversary who queries $\mathsf{MAC}.\mathcal{T}$, $\mathsf{AE}.\mathcal{E}$, $\mathsf{MAC}.\mathcal{V}$, and $\mathsf{AE}.\mathcal{D}$. In what follows, we show how $\mathcal{A}'_{\mathtt{ma}}$ simulates $\mathsf{InitTree}$, Update, and Verify that $\mathcal{A}'$ queries. For $\mathsf{InitTree}$, $\mathcal{A}'_{\mathtt{ma}}$ can simulate it by employing $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ in the same manner as Alg. 14. Note that $\mathsf{InitTree}$ invokes nonce-respecting queries to $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ since $\mathsf{ADD}(u)||\mathsf{CTR}(u) \neq \mathsf{ADD}(u')||\mathsf{CTR}(u')$ holds for all distinct nodes u and u'. In addition, while simulating $\mathsf{InitTree}$, we assume that $\mathcal{A}'_{\mathtt{ma}}$ has two lists to record her queries and responses for $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$, respectively. These lists will be used in the simulation for Verify, hence we will describe the role of them later. Thus, the simulation for $\mathsf{InitTree}$ by $\mathcal{A}'_{\mathtt{ma}}$ is obtained by adding the following operations to Alg. 14.

After line 1: $\mathtt{List}_{\mathsf{AE}} \leftarrow \varepsilon, \mathtt{List}_{\mathsf{MAC}} \leftarrow \varepsilon$

After line 6: $\mathtt{List}_{\mathsf{AE}} \overset{\cup}{\leftarrow} \{(\mathsf{ADD}(\mathsf{leaf}(i)^{\sigma}) \,\|\, \mathsf{CTR}(\mathsf{leaf}(i)^{\sigma}), C[i], \mathsf{Tag}(\mathsf{leaf}(i)^{\sigma}))\}$

After line 9: $\mathtt{List}_{\mathsf{MAC}} \overset{\cup}{\leftarrow} \{(\mathsf{ADD}(u^{\sigma}) \,\|\, \mathsf{CTR}(u^{\sigma}), \mathsf{CTR}(\mathsf{ch}_1(u^{\sigma})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u^{\sigma})), \mathsf{Tag}(u^{\sigma}))\}$

For the above simulation of $\mathsf{InitTree}$, $\mathcal{A}'_{\mathtt{ma}}$ queries $(b^d - 1)/(b-1)$ times to $\mathsf{MAC}.\mathcal{T}$ and queries b^d times to $\mathsf{AE}.\mathcal{E}$.

Next, we show how $\mathcal{A}'_{\mathtt{ma}}$ simulates Update. As well as the proof of privacy for $\mathsf{PAT2}$, $\mathcal{A}'_{\mathtt{ma}}$ can simulate Verify which is a subroutine of Update without querying any oracles since what $\mathcal{A}'_{\mathtt{ma}}$ has to do is only to return $\top$. Regarding other operations of Update except for the subroutine Verify, $\mathcal{A}'_{\mathtt{ma}}$ can simulate them by employing $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ in the same manner as Alg. 16. Also, Update invokes nonce-respecting queries to $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ due to the uniqueness of $\mathsf{ADD}(\cdot)$ and the one-time property of $\mathsf{CTR}(\cdot)$. As well as the simulation for $\mathsf{InitTree}$, $\mathcal{A}'_{\mathtt{ma}}$ records her queries and response of $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$. Thus, the simulation for Update by $\mathcal{A}'_{\mathtt{ma}}$ is obtained by adding the following operations to Alg. 16.

After line 9: $\mathtt{List}_{\mathsf{MAC}} \overset{\cup}{\leftarrow} \{(\mathsf{ADD}(u_i^{\tilde{\sigma}}) \,\|\, \mathsf{CTR}(u_i^{\tilde{\sigma}})\mathsf{CTR}(\mathsf{ch}_1(u_i^{\tilde{\sigma}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\tilde{\sigma}})), \mathsf{Tag}(u_i^{\tilde{\sigma}}))\}$

After line 11: $\mathtt{List}_{\mathsf{AE}} \overset{\cup}{\leftarrow} \{(\mathsf{ADD}(u_d^{\tilde{\sigma}}) \,\|\, \mathsf{CTR}(u_d^{\tilde{\sigma}}), C[idx], \mathsf{Tag}(u_d^{\tilde{\sigma}}))\}$

To simulate q' invocations of Update, $\mathcal{A}'_{\mathtt{ma}}$ queries $q'd$ times to $\mathsf{MAC}.\mathcal{T}$ and queries q' times to $\mathsf{AE}.\mathcal{E}$. Note that the sequence of queries to Update invokes nonce-respecting queries to $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$.

We then show how $\mathcal{A}'_{\mathtt{ma}}$ simulates a query to Verify. In this simulation, $\mathtt{List}_{\mathsf{MAC}}$ and $\mathtt{List}_{\mathsf{AE}}$ are used to prevent $\mathcal{A}'_{\mathtt{ma}}$ from performing *replay queries* to $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$, respectively. A replay query means that $\mathcal{A}'_{\mathtt{ma}}$ queries (N, M, T) (resp. (N, C, T)) to $\mathsf{MAC}.\mathcal{V}$ (resp. $\mathsf{AE}.\mathcal{D}$) while (N, M) (resp. (N, M) such that $(C, T) = \mathsf{AE}.\mathcal{E}(N, M)$) has been queried to $\mathsf{MAC}.\mathcal{T}$ (resp. $\mathsf{AE}.\mathcal{E}$). Such queries may appear in the simulation of unforgeability game since $\mathcal{A}'$ can perform replay attack. Our final goal of this proof is to show how to simulate the unforgeability game that $\mathcal{A}'$ plays using the adversary against MAC $\mathcal{A}_{\mathsf{MAC}}$ and the authenticity adversary against AE $\mathcal{A}_{\mathsf{AE}}$, and then, $\mathcal{A}_{\mathsf{MAC}}$ and $\mathcal{A}_{\mathsf{AE}}$ are prohibited replay queries as defined in Section 2. Therefore, we here have to show $\mathcal{A}'_{\mathtt{ma}}$ can simulate Verify without replay queries. From the above discussion, we define the behavior of $\mathcal{A}'_{\mathtt{ma}}$ when $\mathcal{A}'$ queries (idx', σ') to Verify as follows.

```
Get the path of nodes from the root to the specified leaf, denoted by $(u_0^{\sigma'}, \cdots, u_d^{\sigma'})$. Here, $u_0^{\sigma'}$ is the root node and $u_d^{\sigma'}$ is equal to $\mathsf{leaf}(idx')$.
For $0 \le i \le d-1$, do:
  Set $(N, M, T)$ as $(\mathsf{ADD}(u_i^{\sigma'}) \,\|\, \mathsf{CTR}(u_i^{\sigma'}), \mathsf{CTR}(\mathsf{ch}_1(u_i^{\sigma'})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\sigma'})), \mathsf{Tag}(u_i^{\sigma'})))$.
  If $(N, M, T) \notin \mathtt{List}_{\mathsf{MAC}}$,
    then query to $\mathsf{MAC}.\mathcal{V}$ with $(N, M, T)$ and obtain $\top$ or $\bot$.
    If $\mathsf{MAC}.\mathcal{V}$ returns $\bot$, return $\bot$.
Set $(N, C, T)$ as $(\mathsf{ADD}(u_d^{\sigma'}) \,\|\, \mathsf{CTR}(u_d^{\sigma'}), C[idx], \mathsf{Tag}(u_d^{\sigma'}))$.
If $(N, C, T) \notin \mathtt{List}_{\mathsf{AE}}$,
  then query to $\mathsf{AE}.\mathcal{D}$ with $(N, C, T)$ and obtain $M$ or $\bot$.
  If $\mathsf{AE}.\mathcal{D}$ returns $\bot$, return $\bot$.
Return $\top$.
```

Above definition indicates when $\mathcal{A}'$ invokes replay queries to $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$, $\mathcal{A}'_{\mathtt{ma}}$ can notice it by checking $\mathtt{List}_{\mathsf{MAC}}$ and $\mathtt{List}_{\mathsf{AE}}$, and can simulate the response of $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$ by just returning $\top$. Therefore, $\mathcal{A}'_{\mathtt{ma}}$ can simulate Verify that $\mathcal{A}'$ queries by employing $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$ without performing replay queries to them. For the simulation of Verify, $\mathcal{A}'_{\mathtt{ma}}$ queries at most d times to $\mathsf{MAC}.\mathcal{T}$ and queries at most one time to $\mathsf{AE}.\mathcal{E}$.

Finally, as well as the case of privacy game, the sequence of queries in unforgeability game of $\mathcal{A}'$ invokes nonce-respecting queries to $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ due to the node-unique property of $\mathsf{ADD}(\cdot)$ and the one-time property of $\mathsf{CTR}(\cdot)$. From the above discussions, we observe that $\mathcal{A}'_{\mathtt{ma}}$ querying $\mathsf{MAC}.\mathcal{T}$, $\mathsf{AE}.\mathcal{E}$, $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$ can simulate the functions that $\mathcal{A}'$ queries without performing nonce-repeating queries to $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$, and replay queries to $\mathsf{MAC}.\mathcal{V}$ and $\mathsf{AE}.\mathcal{D}$. We write $\mathcal{A}'^{\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}_{\mathtt{ma}}}$ to describe that $\mathcal{A}'$ queries to $\mathcal{A}'_{\mathtt{ma}}$ pretending to be functions $\mathsf{InitTree}, \mathsf{Update}, \mathsf{Verify}$, and obtain the following equation.

$$\Pr[\mathcal{A}'^{\mathsf{InitTree},\mathsf{Update},\mathsf{Verify}} \to 1] = \Pr[\mathcal{A}'^{\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}_{\mathtt{ma}}} \to 1]. \tag{5}$$

We define new adversary $\mathcal{A}''_{\mathtt{ma}}$ querying $\mathsf{MAC}.\mathcal{T}$, $\mathsf{AE}.\mathcal{E}$, and $\bot_{\mathsf{Tree}}$ so that $\mathcal{A}''_{\mathtt{ma}}$ can simulate functions $\mathsf{InitTree}$, Update, and $\bot_{\mathsf{Tree}}$. Since $\mathcal{A}''_{\mathtt{ma}}$ can simulate $\mathsf{InitTree}$ and Update by employing $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{AE}.\mathcal{E}$ in the same manner as the case that $\mathcal{A}'$ queries to ($\mathsf{InitTree}$, Update, Verify), and $\mathcal{A}''_{\mathtt{ma}}$ only needs to mediate $\mathcal{A}'$'s query to $\bot_{\mathsf{Tree}}$, we obtain the following equation.

$$\Pr[\mathcal{A}'^{\mathsf{InitTree},\mathsf{Update},\bot_{\mathsf{Tree}}} \to 1] = \Pr[\mathcal{A}'^{\mathcal{A}''^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\bot_{\mathsf{Tree}}}_{\mathtt{ma}}} \to 1]. \tag{6}$$

From (5) and (6), we observe

$$\begin{aligned}\mathbf{Adv}^{\mathtt{uftree}}_{\mathsf{Tree}}(\mathcal{A}') &= |\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}''_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{Tree}}}} \to 1]| \\ &\le \underbrace{|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1]|}_{\text{Lemma 3}} \\ &\quad + \underbrace{|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}} \to 1]|}_{\text{Lemma 4}} \\ &\quad + \underbrace{|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}''_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{Tree}}}} \to 1]|}_{\text{Lemma 5}}, \qquad (7)\end{aligned}$$

where $\perp_{\mathsf{AE}}(\cdot,\cdot,\cdot)$ is the function for decryption queries to AE, which always returns $\perp$ for any inputs, and $\perp_{\mathsf{MAC}}(\cdot,\cdot,\cdot)$ is the function for verification queries to MAC, which always returns $\perp$ for any inputs. In the rest of this section, we bound the above three distinguishing probabilities of $\mathcal{A}'$ in (7) in Lemmas 3, 4, and 5.

Lemma 3.

$$|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1]| \le \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{AE}}(\mathcal{A}^{\pm}_{\mathtt{ae}}),$$

where $\mathcal{A}^{\pm}_{\mathtt{ae}}$ is an authenticity adversary against AE *using $b^d + q'$ encryption queries and 1 decryption query.*

Proof. Suppose that $\mathcal{A}'_{\mathtt{ma}}$ eventually outputs a bit in her simulation of unforgeability game and $\mathcal{A}'$ outputs the same bit as $\mathcal{A}'_{\mathtt{ma}}$. Then we obtain the following inequation.

$$\begin{aligned}&|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1]| \\ &\le |\Pr[\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}} \to 1] - \Pr[\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}} \to 1]|.\end{aligned}$$

The rest of the proof is almost the same as that of the privacy bound. We consider $\mathcal{A}'_{\mathtt{ma}}(K_M)$ who owns the MAC key K_M, and obtain the following inequation.

$$\begin{aligned}&|\Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\mathsf{AE}.\mathcal{D}}_{\mathtt{ma}} \to 1] - \Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}_{\mathtt{ma}} \to 1]| \\ &\le |\Pr[\mathcal{A}'_{\mathtt{ma}}(K_M)^{\mathsf{AE}.\mathcal{E},\mathsf{AE}.\mathcal{D}} \to 1] - \Pr[\mathcal{A}'_{\mathtt{ma}}(K_M)^{\mathsf{AE}.\mathcal{E},\perp_{\mathsf{AE}}} \to 1]|. \qquad (8)\end{aligned}$$

The right side of (8) can be seen as the probability that $\mathcal{A}'_{\mathtt{ma}}(K_M)$ querying $\mathsf{AE}.\mathcal{E}$ successfully distinguishes $\mathsf{AE}.\mathcal{D}$ from $\perp_{\mathsf{AE}}$ under the unforgeability game for authentication trees. Without loss of generality, we can assume that this (distinguishing) probability is equal to the probability that $\mathcal{A}'_{\mathtt{ma}}(K_M)$ querying $\mathsf{AE}.\mathcal{E}$ and $\mathsf{AE}.\mathcal{D}$ obtains something other than $\perp$ from $\mathsf{AE}.\mathcal{D}$ under the unforgeability game for authentication trees. Here, let $\mathcal{A}^{\pm}_{\mathtt{ae}}$ be the authenticity adversary against AE. The adversary $\mathcal{A}^{\pm}_{\mathtt{ae}}$ can simulate the oracles that $\mathcal{A}'_{\mathtt{ma}}(K_M)$ queries because the query sequence of $\mathcal{A}'_{\mathtt{ma}}(K_M)$ respects the rule of authenticity game for AE schemes (performing nonce-respecting queries to $\mathsf{AE}.\mathcal{E}$ and not performing replay queries to $\mathsf{AE}.\mathcal{D}$). Thus, we obtain

$$\begin{aligned}&|\Pr[\mathcal{A}'_{\mathtt{ma}}(K_M)^{\mathsf{AE}.\mathcal{E},\mathsf{AE}.\mathcal{D}} \to 1] - \Pr[\mathcal{A}'_{\mathtt{ma}}(K_M)^{\mathsf{AE}.\mathcal{E},\perp_{\mathsf{AE}}} \to 1]| \\ &= |\Pr[\mathcal{A}'_{\mathtt{ma}}(K_M)^{\mathsf{AE}.\mathcal{E},\mathsf{AE}.\mathcal{D}} \text{ forges}]| \\ &= \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{AE}}(\mathcal{A}^{\pm}_{\mathtt{ae}}),\end{aligned}$$

where $\mathcal{A}^{\pm}_{\mathtt{ae}}$ queries $b^d + q'$ times to $\mathsf{AE}.\mathcal{E}$ and queries one time to $\mathsf{AE}.\mathcal{D}$.

Lemma 4.

$$|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}} \to 1]| \le \mathbf{Adv}^{\mathtt{mac}}_{\mathsf{MAC}}(\mathcal{A}_{\mathtt{mac}}),$$

where $\mathcal{A}_{\mathit{mac}}$ is an adversary against MAC *using $(b^d - 1)/(b-1) + q'd$ tagging queries and d verification queries.*

Proof. The proof is almost the same as that of Lemma 3. Firstly, we replace the distinguishing game of $\mathcal{A}'$ to that of $\mathcal{A}'_{\mathtt{ma}}$.

$$\begin{aligned}&|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}} \to 1]|\\ &\leq |\Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}_{\mathtt{ma}} \to 1] - \Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}_{\mathtt{ma}} \to 1]|.\end{aligned}$$

We consider $\mathcal{A}'_{\mathtt{ma}}(K_A)$ who owns AE's key K_A. The probability that $\mathcal{A}'{}_{\mathtt{ma}}$ successfully distinguishes $(\mathsf{MAC}.\mathcal{T}, \mathsf{AE}.\mathcal{E}, \mathsf{MAC}.\mathcal{V}, \perp_{\mathsf{AE}})$ from $(\mathsf{MAC}.\mathcal{T}, \mathsf{AE}.\mathcal{E}, \perp_{\mathsf{MAC}}, \perp_{\mathsf{AE}})$ can be interpreted as the probability that $\mathcal{A}'_{\mathtt{ma}}(K_A)$ querying $\mathsf{MAC}.\mathcal{T}$ and $\mathsf{MAC}.\mathcal{V}$ obtains $\top$ from $\mathsf{MAC}.\mathcal{V}$. Let $\mathcal{A}_{\mathtt{mac}}$ be the adversary against MAC. She can simulate the oracles that $\mathcal{A}'_{\mathtt{ma}}(K_A)$ queries, because the query sequence of $\mathcal{A}'_{\mathtt{ma}}(K_A)$ respects the rule of security game for MAC schemes (performing nonce-respecting queries to $\mathsf{MAC}.\mathcal{T}$ and not performing replay queries to $\mathsf{MAC}.\mathcal{V}$). Thus, we obtain the following inequations.

$$\begin{aligned}&|\Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\mathsf{MAC}.\mathcal{V},\perp_{\mathsf{AE}}}_{\mathtt{ma}} \to 1] - \Pr[\mathcal{A}'^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}_{\mathtt{ma}} \to 1]|\\ &\leq |\Pr[\mathcal{A}'{}_{\mathtt{ma}}(K_A)^{\mathsf{MAC}.\mathcal{T},\mathsf{MAC}.\mathcal{V}} \text{ forges}]|\\ &= \mathbf{Adv}^{\mathtt{mac}}_{\mathsf{MAC}}(\mathcal{A}_{\mathtt{mac}}),\end{aligned}$$

where $\mathcal{A}_{\mathtt{mac}}$ queries $(b^d - 1)/(b-1) + q'd$ times to $\mathsf{MAC}.\mathcal{T}$ and queries d times to $\mathsf{MAC}.\mathcal{V}$.

Lemma 5.

$$|\Pr[\mathcal{A}'^{\mathcal{A}'_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{MAC}},\perp_{\mathsf{AE}}}} \to 1] - \Pr[\mathcal{A}'^{\mathcal{A}''_{\mathtt{ma}}{}^{\mathsf{MAC}.\mathcal{T},\mathsf{AE}.\mathcal{E},\perp_{\mathsf{Tree}}}} \to 1]| = 0.$$

Proof. Both $\mathcal{A}'_{\mathtt{ma}}$ and $\mathcal{A}''_{\mathtt{ma}}$ have the same ways to simulate $\mathsf{InitTree}$ and Update. The adversary $\mathcal{A}'_{\mathtt{ma}}$ simulates verification query of $\mathcal{A}'$ using $\perp_{\mathsf{MAC}}$, $\perp_{\mathsf{AE}}$, $\mathtt{List}_{\mathsf{MAC}}$, and $\mathtt{List}_{\mathsf{AE}}$, while $\mathcal{A}''_{\mathtt{ma}}$ only mediates it to $\perp_{\mathsf{Tree}}$ that returns $\perp$ for any inputs. Namely, the distinguishing probability we evaluate here can be seen as the probability that $\mathcal{A}'$ querying $\mathsf{InitTree}$ and Update obtains $\top$ from the tree verification oracle simulated by $(\perp_{\mathsf{MAC}}, \perp_{\mathsf{AE}})$ with $\mathtt{List}_{\mathsf{MAC}}$ and $\mathtt{List}_{\mathsf{AE}}$. This probability is equal to the probability that the data associated with nodes in the path verified in the verification query of $\mathcal{A}'$ consists only of the data in $\mathtt{List}_{\mathsf{MAC}}$ and $\mathtt{List}_{\mathsf{AE}}$ (see line 4 and 8 of the simulation of Verify in the proof of Theorem 4). In the following claim, we prove the probability is equal to zero.

Claim. Recall that (idx', σ') is the tree verification query and $(u_0, \cdots, u_d)$ is the path of nodes from the root to the specified leaf, then either **(a)** or **(b)** described below must hold.

(a) There exists $i \in \{0, \ldots, d{-}1\}$ such that $(\mathsf{ADD}(u_i^{\sigma'}) \,\|\, \mathsf{CTR}(u_i^{\sigma'}), \mathsf{CTR}(\mathsf{ch}_1(u_i^{\sigma'})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\sigma'})), \mathsf{Tag}(u_i^{\sigma'})) \notin \mathtt{List}_{\mathsf{MAC}}$.

(b) $(\mathsf{ADD}(u_d^{\sigma'}) \,\|\, \mathsf{CTR}(u_d^{\sigma'}),\, C^{\sigma'}[idx],\, \mathsf{Tag}(u_d^{\sigma'})) \notin \mathtt{List}_{\mathsf{AE}}$.

This claim states that $\mathcal{A}'_{\mathtt{ma}}$ querying $\mathsf{MAC}.\mathcal{T}$, $\mathsf{AE}.\mathcal{E}$, $\perp_{\mathsf{MAC}}$ and $\perp_{\mathsf{AE}}$ has to query to $\perp_{\mathsf{MAC}}$ or $\perp_{\mathsf{AE}}$ in her simulation of the tree verification query[15]. Thus, she always obtains $\perp$ from $\perp_{\mathsf{MAC}}$ or $\perp_{\mathsf{AE}}$ and she returns it to $\mathcal{A}'$.

All that remains is the proof of the claim. If **(b)** occurs, the claim is simply proved. We need to see that if **(b)** does not occur, **(a)** must hold. First we discuss u_0 (*i.e.*, the root node). Let $(N_{u_0}, M_{u_0}, T_{u_0}) = (\mathsf{ADD}(u_0^{\sigma'}) \,\|\, \mathsf{CTR}(u_0^{\sigma'}),\, \mathsf{CTR}(\mathsf{ch}_1(u_0^{\sigma'})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_0^{\sigma'})), \mathsf{Tag}(u_0^{\sigma'}))$. If $(N_{u_0}, M_{u_0}, T_{u_0}) \notin \mathtt{List}_{\mathsf{MAC}}$ holds, it means that **(a)** holds. Suppose that $(N_{u_0}, M_{u_0}, T_{u_0}) \in \mathtt{List}_{\mathsf{MAC}}$, and we obtain the following equation.

$$(N_{u_0}, M_{u_0}, T_{u_0}) = (\mathsf{ADD}(u_0^{\tilde{\sigma}_{q'}}) \,\|\, \mathsf{CTR}(u_0^{\tilde{\sigma}_{q'}}), \mathsf{CTR}(\mathsf{ch}_1(u_0^{\tilde{\sigma}_{q'}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_0^{\tilde{\sigma}_{q'}})), \mathsf{Tag}(u_0^{\tilde{\sigma}_{q'}})), \qquad (9)$$

[15] The literature [HJ06] shows almost the same claim and its proof for proposed authentication tree without encryption of leaf nodes, however the proof is a little bit complex. We recast it for the sake of ease to understand and the authentication tree with encryption of leaf nodes.

which means that the data stored in $u_0^{\sigma'}$ is the same as that stored in $u_0^{\tilde{\sigma}_{q'}}$. This holds because N_{u_0} cannot be tampered by definition, and the element of $\mathtt{List}_{\mathsf{MAC}}$ including N_{u_0} is uniquely determined as (9) since nonces included in $\mathtt{List}_{\mathsf{MAC}}$ are distinct. Note that we also obtain $\mathsf{CTR}(u_1^{\sigma'}) = \mathsf{CTR}(u_1^{\tilde{\sigma}_{q'}})$ from (9).

Next, we discuss u_1. Let $(N_{u_1}, M_{u_1}, T_{u_1}) = (\mathsf{ADD}(u_1^{\sigma'}) \,\|\, \mathsf{CTR}(u_1^{\sigma'}), \mathsf{CTR}(\mathsf{ch}_1(u_1^{\sigma'})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_1^{\sigma'})), \mathsf{Tag}(u_1^{\sigma'}))$. As well as the case of u_0, we can suppose that $(N_{u_1}, M_{u_1}, T_{u_1}) \in \mathtt{List}_{\mathsf{MAC}}$ since **(a)** occurs when $(N_{u_1}, M_{u_1}, T_{u_1}) \notin \mathtt{List}_{\mathsf{MAC}}$ holds. In the same manner as the case of u_0, we obtain

$$(N_{u_1}, M_{u_1}, T_{u_1}) = (\mathsf{ADD}(u_1^{\tilde{\sigma}_{q'}}) \,\|\, \mathsf{CTR}(u_1^{\tilde{\sigma}_{q'}}), \mathsf{CTR}(\mathsf{ch}_1(u_1^{\tilde{\sigma}_{q'}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_1^{\tilde{\sigma}_{q'}})), \mathsf{Tag}(u_1^{\tilde{\sigma}_{q'}})),$$

since $\mathsf{ADD}(u_1^{\sigma'})$ cannot be tampered and $\mathsf{CTR}(u_1^{\sigma'}) = \mathsf{CTR}(u_1^{\tilde{\sigma}_{q'}})$ holds from (9).

Suppose that we repeat the same discussion as u_0 and u_1. For $2 \leq i \leq d-1$, we define $(N_{u_i}, M_{u_i}, T_{u_i}) = (\mathsf{ADD}(u_i^{\sigma'}) \,\|\, \mathsf{CTR}(u_i^{\sigma'}), \mathsf{CTR}(\mathsf{ch}_1(u_i^{\sigma'})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\sigma'})), \mathsf{Tag}(u_i^{\sigma'}))$. When $(N_{u_i}, M_{u_i}, T_{u_i}) \in \mathtt{List}_{\mathsf{MAC}}$ for $0 \leq i \leq d-1$, we obtain the following equation.

$$(N_{u_i}, M_{u_i}, T_{u_i}) = (\mathsf{ADD}(u_i^{\tilde{\sigma}_{q'}}) \,\|\, \mathsf{CTR}(u_i^{\tilde{\sigma}_{q'}}), \mathsf{CTR}(\mathsf{ch}_1(u_i^{\tilde{\sigma}_{q'}})) \,\|\, \cdots \,\|\, \mathsf{CTR}(\mathsf{ch}_b(u_i^{\tilde{\sigma}_{q'}})), \mathsf{Tag}(u_i^{\tilde{\sigma}_{q'}})). \quad (10)$$

Finally, we discuss u_d (*i.e.*, the leaf node). Let $(N_{u_d}, C_{u_d}, T_{u_d}) = (\mathsf{ADD}(u_d^{\sigma'}) \,\|\, \mathsf{CTR}(u_d^{\sigma'}),\ C^{\sigma'}[idx], \mathsf{Tag}(u_d^{\sigma'}))$. Recall that we assumed that **(b)** did not occur, thus $(N_{u_d}, C_{u_d}, T_{u_d}) \in \mathtt{List}_{\mathsf{AE}}$. Here, $\mathsf{ADD}(u_d^{\sigma'}) = \mathsf{ADD}(u_d^{\tilde{\sigma}'_q})$ holds since $\mathsf{ADD}(\cdot)$ cannot be tampered, and $\mathsf{CTR}(u_d^{\sigma'}) = \mathsf{CTR}(u_d^{\tilde{\sigma}'_q})$ holds due to (10) when $i = d-1$. Thus, we have

$$(N_{u_d}, C_{u_d}, T_{u_d}) = (\mathsf{ADD}(u_d^{\tilde{\sigma}_{q'}}) \,\|\, \mathsf{CTR}(u_d^{\tilde{\sigma}_{q'}}), C^{\tilde{\sigma}_{q'}}[idx], \mathsf{Tag}(u_d^{\tilde{\sigma}_{q'}})), \quad (11)$$

since nonces included in $\mathtt{List}_{\mathsf{AE}}$ are distinct, hence the element of $\mathtt{List}_{\mathsf{AE}}$ including N_{u_d} is uniquely determined as (11).

From (10) and (11), we proved that the data stored in $u_i^{\sigma'}$ is the same as that stored in $u_i^{\tilde{\sigma}_{q'}}$ for all $i \in \{0, \ldots, d\}$, which is a forbidden query. Therefore, there must exist $i \in \{0, \ldots, d-1\}$ such that $(N_{u_i}, M_{u_i}, T_{u_i}) \notin \mathtt{List}_{\mathsf{MAC}}$. This concludes the claim.

From (7), Lemmas 3, 4, and 5, we obtain

$$\mathbf{Adv}^{\mathtt{uftree}}_{\mathsf{Tree}}(\mathcal{A}') \leq \mathbf{Adv}^{\mathtt{auth}}_{\mathsf{AE}}(\mathcal{A}^{\pm}_{\mathtt{ae}}) + \mathbf{Adv}^{\mathtt{mac}}_{\mathsf{MAC}}(\mathcal{A}_{\mathtt{mac}}),$$

where $\mathcal{A}^{\pm}_{\mathtt{ae}}$ is the authenticity adversary against AE and $\mathcal{A}_{\mathtt{mac}}$ is the adversary against MAC. The adversary $\mathcal{A}^{\pm}_{\mathtt{ae}}$ queries $b^d + q'$ times to the encryption oracle and queries one time to the decryption oracle. The adversary $\mathcal{A}_{\mathtt{mac}}$ queries $(b^d - 1)/(b-1) + q'd$ times to the tagging oracle and queries d times to the verification oracle.

6 Implementation and Evaluation

In this section, we demonstrate the hardware implementation of the proposed scheme and evaluate it using logic synthesis. We instantiate our scheme using AES as the block cipher, which indicates that $n = 128$. We assume that the lengths of the memory address and the counter are both 64 bits (*i.e.*, $\alpha = \beta = n/2$). The tag is given by 64 bits (*i.e.*, $\tau = 64$), which corresponds to the security level of SIT (*i.e.*, the bit length of keys in the inner-product MAC in [Gue16a]) for a fair comparison. In this paper, we focus on a high-throughput and area-time efficient architecture based on an unrolled and pipelined AES datapath, similar to that of SIT in [Gue16a], whose throughput is one block encryption per clock cycle. This high-throughput architecture suits the context of memory encryption. Note that our architecture can utilize other block ciphers and architectures (*e.g.*, round-based and byte-serial ones) in accordance with the optimization goals. (The variations are discussed in the next section.)

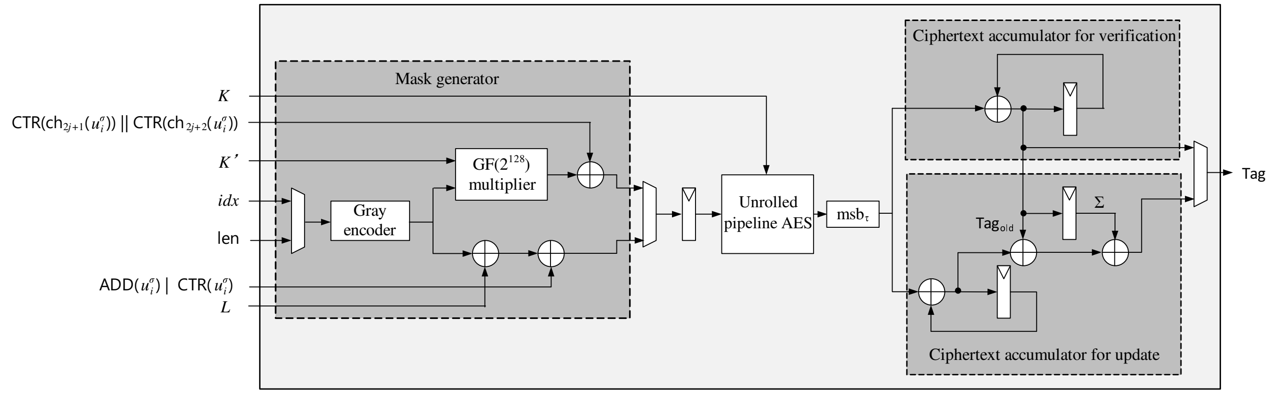

Mask generator
K
CTR(ch$_{2j+1}$(u_i^σ)) || CTR(ch$_{2j+2}$(u_i^σ))
K'
idx
len
ADD(u_i^σ) | CTR(u_i^σ)
L
Gray encoder
GF(2^{128}) multiplier
Unrolled pipeline AES
msb$_\tau$
Ciphertext accumulator for verification
Ciphertext accumulator for update
Tag$_{\text{old}}$
Σ
Tag

Fig. 6: Proposed MAC hardware architecture.

6.1 MAC Hardware Architecture

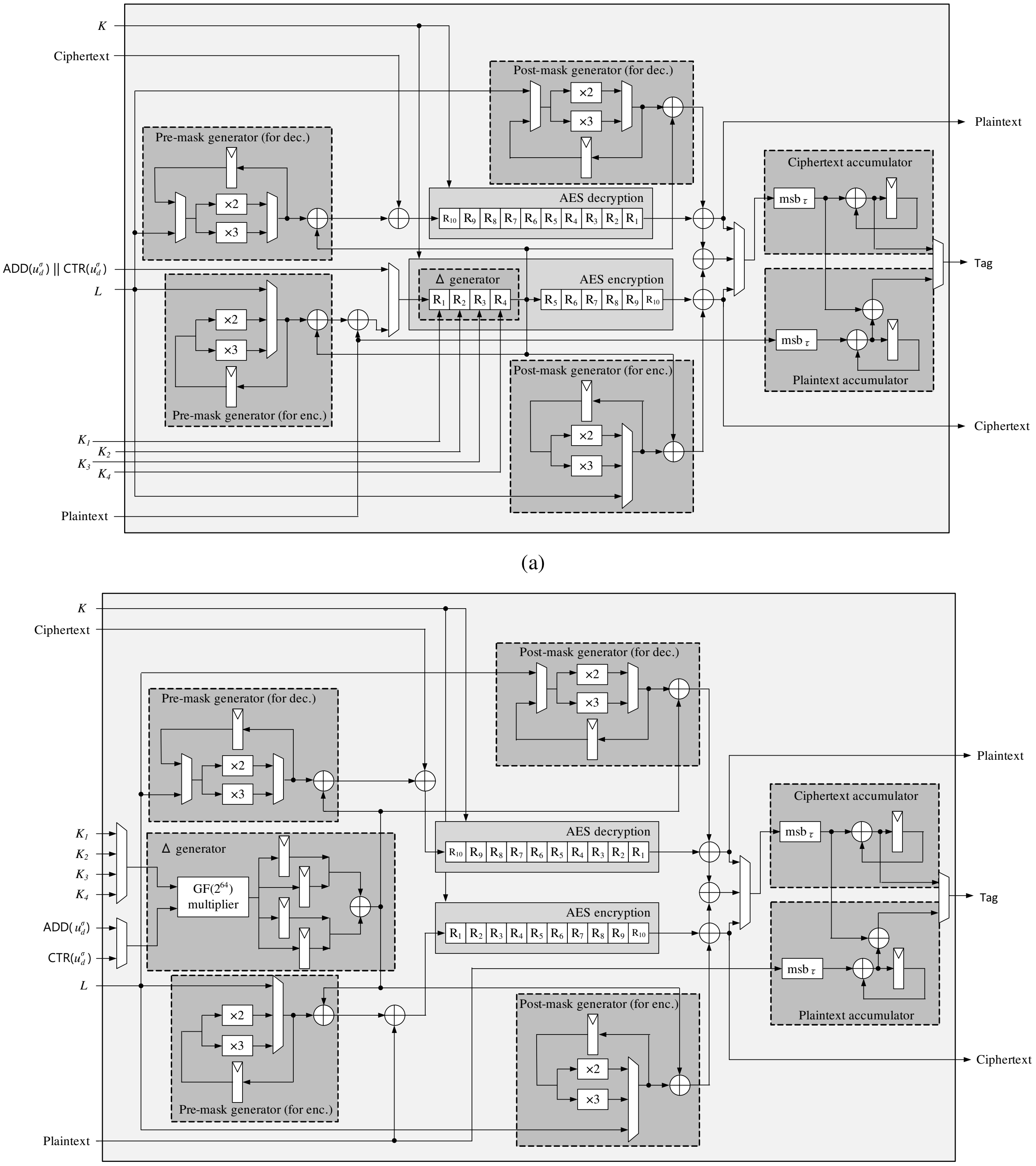
Figure 6 shows the proposed hardware architecture of PXOR-MAC. The primary inputs consist of a block index, the number of branches len (=b), nonce ($\mathsf{ADD}(u_i^\sigma) \,\|\, \mathsf{CTR}(u_i^\sigma)$), two n-bit keys, and an n-bit segmented plaintext block ($\mathsf{CTR}(\mathsf{ch}_{2j+1}(u_i^\sigma)) \,\|\, \mathsf{CTR}(\mathsf{ch}_{2j+2}(u_i^\sigma))$) ($0 \leq j \leq b/2-2$), and the primary output is given as tag. One plaintext block is fed to the hardware every clock cycle one after another and an encoding is completed with 11 clock cycles. In this architecture, the AES datapath is fully unrolled and pipelined. The pipeline registers are inserted at the boundaries of each round in order to increase the throughput. This enables the encryption of one plaintext block in one clock cycle with the frequency corresponding to the critical path of one round datapath.

An up-to-date AES round datapath with a tower-field S-box presented in [UMM$^+$20] is adopted for ELM (and SIT [Gue16a] for a comparison in this paper) in the following hardware implementation. A mask value for the input block to the AES core ($K' \cdot i$ in Alg. 13) is generated by the multiplication of a gray code (converted from a block index) and a key K' over GF(2^n). The conversion from a block index to a gray code is given by a combinational circuit and the generation of a mask value is implemented using a ($n \times \log b$)-bit GF multiplier. This multiplier generates mask values from all indices in a tree with b branches in one clock cycle. The mask value for the nonce ($N_{\text{old}} \oplus K' \cdot m \oplus L$ in Alg. 13) is computed as the sum of the last mask value and L without any GF multiplication. The accumulation in the tag generator is implemented by a feedback loop consisting of a bit-parallel-XOR (*i.e.*, GF($2^{n/2}$) adder) and registers, which realizes the for loop at lines 3–9 in Alg. 13.

During PXOR-MAC.$\mathcal{V}$ for Verify and PXOR-MAC.$\mathcal{VU}$ for Update, the tag is computed using the upper ciphertext accumulator after the encryption result is truncated into τ bits at msb$_\tau$. The accumulator is given by a feedback loop with a bit-parallel-XOR (*i.e.*, $GF(2^{n/2})$ adder). When updating the tag, the intermediate value Σ as mentioned in Alg. 13 computed in the pre-verification process is stored into a register in the lower ciphertext accumulator for the following update procedure to exploit the incremental property of PXOR-MAC. The lower feedback loop after msb$_\tau$ is used for the accumulation to compute Σ. The updated tag is then calculated by XORing Σ and the old tag T_{old} computed in pre-verification.

6.2 AE Hardware Architecture

Figure 7 shows the hardware architectures of the proposed AEs, where (a) and (b) show the architectures using AES4 and GF($2^{n/2}$) multiplication (*i.e.*, Flat-OCB-f and Flat-OCB-m), respectively. The proposed architectures utilize one encryption core and one decryption core. Both cores are unrolled and pipelined similarly to the above MAC hardware to ensure high throughput. The encryption and decryption cores are separately implemented (without unifying them like [UMM$^+$20, UMHA16]) in order to perform a decryption in


K
Ciphertext
Post-mask generator (for dec.)
×2
×3
Pre-mask generator (for dec.)
AES decryption
R10 R9 R8 R7 R6 R5 R4 R3 R2 R1
Plaintext
Ciphertext accumulator
msb τ
ADD(u_d^σ) || CTR(u_d^σ)
L
Δ generator
AES encryption
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10
Tag
Plaintext accumulator
Post-mask generator (for enc.)
Pre-mask generator (for enc.)
K_1
K_2
K_3
K_4
Ciphertext
Plaintext
ADD(u_d^σ)
CTR(u_d^σ)
GF(2^{64}) multiplier

(a)

(b)

Fig. 7: Proposed AE hardware architecture for (a) Flat-OCB-f and (b) Flat-OCB-m.

the pre-verification process and an encryption in the update process simultaneously for Update. The pre-mask and post-mask generators compute the mask values for the input and output of encryption/decryption, respectively. The proposed architecture utilizes two pre-mask generators and two post-mask generators for simultaneous encryption and decryption. The field doubling and tripling for mask value generation are achieved by combinational circuit blocks denoted by ×2 and ×3, which consist of four and 132 two-way XOR gates, respectively. Two pre-mask generators and two post-mask generators can be implemented with less area than another utilization consisting of one pre-mask generator, one post-mask generator, and two 128-bit-wise first-in first-out (FIFO) buffers[16]. Plaintext accumulators obtain truncated plaintext blocks for generating the tag, which consists of a feedback loop with a bit-parallel-XOR (*i.e.*, $\mathrm{GF}(2^{n/2})$ adder). Finally, $\widetilde{E}_K^{N,0,0}(0^n)$ is added before outputting the tag.

The mask value $\mathit{\Delta}$ is generated from a nonce by the $\mathit{\Delta}$ generator module. In $\mathsf{Flat\text{-}OCB\text{-}f}$ shown in Fig. 7(a) (*i.e.*, the proposed AE with $\mathsf{MASK1}$ in Alg. 8), $\mathit{\Delta}$ is generated by the four round datapaths of the above-mentioned unrolled-pipelined AES using four distinct 128-bit keys K_1, K_2, K_3 and K_4 as round keys. In $\mathsf{Flat\text{-}OCB\text{-}m}$ at Fig. 7(b) (*i.e.*, the proposed AE with $\mathsf{MASK2}$ in Alg. 9), $\mathit{\Delta}\ (= (N_1 \cdot K_1 \parallel N_2 \cdot K_2) \oplus (N_2 \cdot K_3 \parallel N_1 \cdot K_4))$ is computed using a $((n/2) \times (n/2))$-bit GF multiplier with four clock cycles, where four distinct $(n/2)$-bit secret keys K_1, K_2, K_3, and K_4 are used. The generated $\mathit{\Delta}$ is added to the mask values at the pre/post-mask generators.

For Verify, the proposed architecture first computes $\mathit{\Delta}$. The generated $\mathit{\Delta}$ is added to doubled or tripled L at the pre-mask generator (for dec.), and then the resulting mask value $\mathit{\Delta} \oplus 2L$ or $\mathit{\Delta} \oplus 3L$ is added to the input ciphertext block before decryption. After performing the decryption, the mask value from the post-mask generator (for dec.) is added to the decryption result to obtain the corresponding plaintext. At the same time as the first block is being decrypted, $\widetilde{E}_K^{N,0,0}(0^n)$ is computed using the encryption core and then added to the above plaintext. The resulting value is truncated into τ bits and is stored in the register in the plaintext accumulator for the verification. Then, the second and subsequent blocks are processed in parallel in the pipelined datapath, and the processed blocks are accumulated in the plaintext accumulator. After processing all blocks, the architecture outputs the verification tag.

For Update, the proposed architecture performs a pre-verification and an update processes simultaneously thanks to the separately-implemented decryption and encryption cores. Let $\mathit{\Delta}_{\mathrm{old}}$ and $\mathit{\Delta}_{\mathrm{new}}$ be the parts of the mask values generated from a nonce for the pre-verification and update processes, respectively. Initially, $\mathit{\Delta}_{\mathrm{old}}$ is first generated and then $\mathit{\Delta}_{\mathrm{new}}$ is generated using the $\mathit{\Delta}$ generator module. In the case of $\mathsf{AES4}$ in Fig. 7(a), the generations of $\mathit{\Delta}_{\mathrm{old}}$ and $\mathit{\Delta}_{\mathrm{new}}$ are executed in parallel owing to the pipelined datapath. In contrast, in the case of the $\mathrm{GF}(2^{n/2})$ multiplier in Fig. 7(b), the multiplication results of $N_1 \cdot K_1$ and $N_1 \cdot K_4$ are reused because the value of $\mathsf{ADD}(u^\sigma)$ (*i.e.*, half of the nonce) is fixed in the pre-verification and update processes, which means that $N_1 \cdot K_1$ and $N_1 \cdot K_4$ are identical for the pre-verification and update processes. Therefore, the number of clocks can be reduced to two from four for computation of $\mathit{\Delta}_{\mathrm{new}}$. Then, the architecture generates the tag verification in the same manner as Verify. In addition, after computing $\widetilde{E}_K^{N_{\mathrm{old}},0,0}(0^n)$, we simultaneously compute $\widetilde{E}_K^{N_{\mathrm{new}},0,0}(0^n)$, encrypt the plaintext blocks, and accumulate the results at the plaintext accumulator module (for the update process). After processing all plaintext blocks, the architecture outputs the updated tag.

6.3 Performance Evaluation

This subsection reports our performance evaluation of the proposed architectures and SIT, a major state-of-the-art counterpart, on the basis of logic synthesis results. We assume that one MAC module (hardware) is used at the top and each middle layer in a tree structure and one AE module is used at the lowest layer. In other words, an authentication tree with a depth of d utilizes d MAC modules and one AE module, which fully exploits the parallelism provided by the tree structure as the claim of PAT. Under this assumption, we

[16] The mask value generated by the pre-mask generator should be retained for ten clock cycles for post-mask addition. This indicates that we require a (128 × 10)-bit register to implement one FIFO if we use AES as the block cipher, which consumes a larger area and power than the four mask generators in our architecture.

Table 2: Circuit areas synthesized with 4GHz timing constraint [GE]

depth d	SIT	ELM1	ELM2
3	421,083.25	502,701.12	519,910.50
5	601,544.50	700,342.25	717,801.00
7	914,155.25	898,443.25	915,612.50

investigate the best-case performance given a constraint in area and power (*i.e.*, the number of available MAC modules), and clarify the area-latency trade-offs from the performance evaluation results.

For the evaluation, we use Synopsys Design Compiler I-2013.12-SP5 and Nangate 15nm Open Cell Library. The performance of authentication trees with $d = 3$, 5, and 7 are evaluated as the major parameter values as in [Gue16a]. Table 2 lists the area obtained from the logic synthesis. ELM1 and ELM2 represent the proposed schemes where Flat-OCB-f and Flat-OCB-m are used as the AE, respectively. We set the timing-constraint for the synthesis to the operating frequency of 4GHz, assuming that the authentication tree is deployed for memory encryption in modern high-end CPUs operating at 3GHz or faster. We confirmed that no timing violation was found in the synthesis result, and therefore the proposed architecture can be used even for modern high-end CPUs without degrading the system clock frequency.

For comparison, Table 2 also lists the synthesis results of SIT implemented under the same conditions and assumption as above. The SIT was implemented according to [Gue16a]. We utilized the unrolled-pipelined AES encryption core with the same round datapath as our scheme. A 64 × 64-bit GF multiplier to compute the inner-product MAC was also implemented in the same manner as ours. Therefore, the critical path was given by the one round datapath of AES, like ours. The SIT also utilized $d + 1$ modules when the depth was d, as the AE and MAC of SIT are given by the same module (*i.e.*, an AES encryption core and a GF(2^{64}) multiplier).

From Table 2, we can see that the area of the proposed architecture can be comparable with that of the SIT as the depth was larger. The hardware architectures for Flat-OCB require both encryption and decryption cores, which resulted in a larger area than the inverse-free AE used in the SIT. However, the proposed MAC hardware is implemented using only one AES encryption core as the major component, whereas the SIT requires a GF(2^{64}) multiplier in addition to one AES encryption core. Consequently, we can confirm that the proposed authentication trees have an advantage over the SIT in terms of latency and memory regions for the cases with large depths.

Table 3 shows the numbers of clock cycles (*i.e.*, latency) and the size of protected memory region (namely, the covered region) of ELM and SIT for various tree parameters. The corresponding comparison graphs for major parameters are shown in Fig. 8. Incremental SIT (Incr. SIT for short) indicates the evaluation result of SIT when Update is performed in an incremental manner. Note that such an incremental update has not previously been mentioned in the literature [Gue16a]. We also evaluate the corresponding incremental SIT for a fair comparison. Each clock cycle shown here is given by a larger one of either AE or MAC. For example, if our authentication tree has eight branches $b = 8$ and handles 512-bit blocks $\ell = 512$ [bit] at the leaf (or lowest) node, ELM1 requires 18 and 24 clock cycles for MAC and AE, respectively; hence the clock cycle of this authentication tree is given as 24 clock cycles in the table. The bold-face characters in each row highlight the scheme that achieved the lowest latency (minimal clock cycles) under the parameter condition of the row. The parameters used in Fig. 8 are underlined in the chunk size column in Table 3. The table shows the results of all tree structures comprehensively, where some rows hatched in gray indicate better clock cycles than those in white in terms of the latency required for the covered regions. For example, a tree with $b = 8$ and $\ell = 4{,}096$ requires a larger latency and a smaller covered region than that with $b = 16$ and $\ell = 512$, and therefore there is no reason to use the former tree rather than the latter. Such meaningless parameters are caused by the gap in latency between AE and MAC, as discussed in Section 6.1.3.

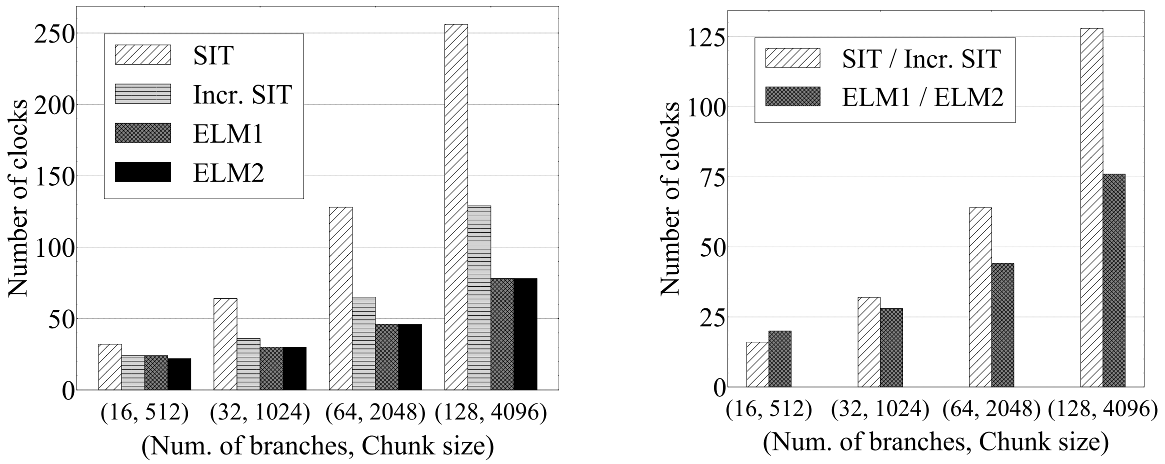
Table 3 and Fig. 8 show that the advantage of the proposed scheme (ELM1 and ELM2) is greater as the covered region becomes larger. One major reason is that the proposed scheme utilizes a 128-bit block cipher (*i.e.*, AES), whereas the SIT processes a plaintext in a 64-bit-wise manner (*i.e.*, inner-product MAC over

Table 3: Numbers of clocks to update and verify tag.

		Update				Verify		Covered region [Byte]		
b	ℓ	SIT	Incr. SIT	ELM1	ELM2	SIT / Incr. SIT	ELM	$d=3$	$d=5$	$d=7$
8	512	**20**	**20**	24	21	**14**	18	32K	2M	134M
	1,024	32	32	28	**25**	**18**	22	65K	4M	268M
	2,048	64	64	36	**33**	32	**30**	131K	8M	536M
	4,096	128	128	52	**49**	64	**46**	262K	16M	1G
	8,192	256	256	84	**81**	128	**78**	524K	33M	2G
16	512	32	**20**	24	22	**16**	20	262K	67M	17G
	1,024	32	32	28	**25**	**18**	22	524K	134M	34G
	2,048	64	64	36	**33**	32	**30**	1M	268M	68G
	4,096	128	128	52	**49**	64	**46**	2M	536M	137G
	8,192	256	256	84	**81**	128	**78**	4M	1G	274G
32	512	64	33	**30**	**30**	32	**28**	2M	2G	2T
	1,024	64	33	**30**	**30**	32	**28**	4M	4G	4T
	2,048	64	64	36	**33**	32	**30**	8M	8G	8T
	4,096	128	128	52	**49**	64	**46**	16M	17G	17T
	8,192	256	256	84	**81**	128	**78**	33M	34G	35T
64	512	128	65	**46**	**46**	64	**44**	16M	68G	281T
	1,024	128	65	**46**	**46**	64	**44**	33M	137G	562T
	2,048	128	65	**46**	**46**	64	**44**	67M	274G	1P
	4,096	128	128	52	**49**	64	**46**	134M	549G	2P
	8,192	256	256	84	**81**	128	**78**	268M	1T	4P
128	512	256	129	**78**	**78**	128	**76**	134M	2T	36P
	1,024	256	129	**78**	**78**	128	**76**	268M	4T	72P
	2,048	256	129	**78**	**78**	128	**76**	536M	8T	144P
	4,096	256	129	**78**	**78**	128	**76**	1G	17T	288P
	8,192	256	256	84	**81**	128	**78**	2G	35T	576P

GF(2^{64})). More precisely, since the MAC module at each layer (and AE module) should process more bits for a larger parameter, the 128-bit-wise computation of PXOR-MAC in the proposed scheme enables fewer calls of the underlying pseudo-random function than the 64-bit-wise computation of the SIT, which results in a lower latency of the proposed scheme. In addition, the number of clock cycles in the update process of AE is reduced by using a distinct decryption core to perform the pre-verification and update processes simultaneously. Note that the SIT uses an Encrypt-then-MAC for AE, which is given by the counter-mode encryption followed by inner-product MAC. Since SIT does not utilize any decryption function and the MAC computation becomes critical for the latency, the latency of SIT cannot be reduced in the same manner as our scheme. In contrast, for small parameters, such as $b = 8$ (or $b = 16$) and $\ell = 512$ (which is the original parameter for SIT in [Gue16a]), the SIT has a lower latency for pre-verification and update processes thanks to the lightness of GF(2^{64}) multiplication. These results suggest that ELM is superior to the SIT for most of the parameters, especially when covering a larger region. As the memory region to be protected becomes larger, the advantage of ELM increases significantly.

The on-chip and off-chip memory sizes for each architecture are listed in Table 4. We assume that both counter and tag lengths are 56 bits for comparison with SIT. With respect to the on-chip storage, ELM1 requires four 128-bit round keys for $\mathit{\Delta}$ gen., a 128-bit L, a 128-bit plaintext/ciphertext processing key K, and a 56-bit $\mathsf{CTR}(u^{\sigma}_{\mathsf{root}})$. Similarly, ELM2 requires four 64-bit round keys for $\mathit{\Delta}$ gen., a 128-bit L, a 128-bit plaintext/ciphertext processing key K, and a 56-bit $\mathsf{CTR}(u^{\sigma}_{\mathsf{root}})$. Here, the amount of on-chip storage in ELM1 and ELM2 is constant regardless of parameters b and ℓ for the tree. In contrast, the SIT needs to store the



(a) Update

(b) Verify

Fig. 8: Typical numbers of clock cycles to update and verify tag.

Table 4: Required memory sizes

	SIT	ELM1	ELM2
on-chip	$320 + \max\{\ell, 64b\}$	1,208	952
off-chip	$113 \times \sum_{i=0}^{d} b^i - 56$	$112 \times \sum_{i=0}^{d} b^i - 56$	$112 \times \sum_{i=0}^{d} b^i - 56$

2×128-bit keys and inner-product MAC for nonce processing and mask generation, depending on the size of the tree parameters. As a result, the on-chip memory size gets larger when the tree parameters b and ℓ are larger because the key length of inner-product MAC increases in proportion to the length of the input block(*i.e.*, b). For example, the on-chip memory size is 768 bits when $b = 8$ and $\ell = 512$, whereas it is 8,448 bits when $b = 128$ and $\ell = 8{,}192$.

In addition, the off-chip memory size is determined by the size of the counters excluding the root and tags. In the SIT, the counters at $d = 1$ are originally stored in the on-chip memory, but here we store it in the off-chip memory for comparison. Our tree requires CTR and Tag for the middle/lowest layers, and Tag for the root layer, all of which are given in 56 bit. Because SIT stores one unused bit in each layer, the size of its memory unit is one bit larger than that of ours. For example, when $d = 3$ and $b = 8$, the off-chip memory sizes of SIT and ours are 66,049 and 65,464 bits, respectively.

7 Discussion

7.1 Design Optimizations

Considering System Constraints. In this paper, we evaluated the performance of our authentication trees without considering system constraints in order to demonstrate the scalability of the proposed scheme. In practice, we need to design and optimize the total hardware configuration depending on various system/architecture constraints including system clock frequency, memory size to be protected, available resource in the on-chip size, memory bandwidth, cache memory structure, and so on. The sizes of the MAC and AE modules should also be determined considering the above constraints, though here we used d MAC modules and one AE module for the authentication tree implementation according to the parallelizability of the authentication tree. (Note again that PAT was proposed as the first scheme that offers such a parallelizability.)

For example, as for the operation frequency, the result discussed in Section 6.3 suggests that our architectures should not limit the operating frequency since the system clock frequency of modern high-end CPUs is currently at most 3.8GHz unless overclocking occurs (*e.g.*, Intel Core i7-10700K and AMD Ryzen 3900X)[17]. When the maximum frequency of the MAC and AE modules is far higher than the system clock, and if it is allowed under the system constraints, the number of clock cycles for encryption and decryption can be reduced without changing the system clock frequency by removing and rearranging the pipeline registers in the unrolled AES datapath appropriately. In contrast, when the system clock frequency is higher than the maximum frequency of MAC and AE modules, we should modify the datapath to enhance the frequency for the deployment.

Mitigating Gap in Latency between AE and MAC. A gap in latency between AE and MAC leads to a loss of efficiency for the authentication tree because the entire latency is determined by a larger latency of either AE or MAC. (This gap is the reason most rows in Table 3 are denoted in gray. Only well-balanced parameters are meaningful.) While we systematically evaluated the typical tree structures in Section 6.3, where the parameters are given by the power of two, these parameters should be determined such that the latencies of AE and MAC are well-balanced.

However, it would be difficult to align the latencies of AE and MAC exactly. In such a case, it is effective to reduce the number of pipeline stages of AES in either AE or MAC with larger latency. In addition, selecting the appropriate S-box implementation would be useful, as the above evaluation utilized a tower-field S-box for achieving high area-time efficiency [UMM$^+$20]. Since the AES encryption/decryption core was unrolled and pipelined, the usage of a table-based S-box for two consecutive rounds makes it possible to remove the pipeline register between them (*i.e.*, reduce the number of clock cycles) without significantly degrading the operation frequency. In other words, two rounds can be computed in one clock cycle if we use a table-based S-box for the rounds. We confirmed through the logic synthesis with NanGate 15nm Open Cell Library that such an implementation could operate at 4GHz.

Here, our architecture for Flat-OCB-f (*i.e.*, proposed AE with AES4) uses an AES encryption core for the generation of Δ from nonce and the encryption of plaintext blocks. It is particularly effective to reduce the latency of the four-round datapath used for AES4 using table-based S-box—because, in Flat-OCB-f, the four round datapath is used for both nonce processing and plaintext encryption.

In summary, when designing the authentication tree and its hardware, we should first determine the optimal (*i.e.*, well-balanced) tree structure parameters for the required covered region. Then, we can mitigate the remaining gap in latency between AE and MAC on the basis of the above hardware optimization approach.

7.2 Application of Split Counter

The split counter is a method to reduce the amount of counters stored in an off-chip for memory authentication trees [YEP$^+$06]. It uses two types of counters: major and minor ones. A major counter is shared by the children nodes of the node of interest (or a parent node), and each child node is equipped with a minor counter. In other words, in an authentication tree with split counter, children nodes having the same parent node share the upper bits of the same major counter.

ELM can also be applied to the split counter. In the following, we evaluate the off-chip memory size for the case where the split counter is used. We assume here that both tag and counter lengths without the split counter are 64 bits and the major and minor counters are given as 56 and 8 bits, respectively. The off-chip memory size of the entire tree is $128 \times \sum_{i=0}^{d} b^i - 64$ and $72 \times \sum_{i=0}^{d} b^i - 64 + 56 \times \sum_{i=0}^{d-1} b^i + 56$ without and with the split counter, respectively. The value of 72 is the sum of the tag and minor counter lengths, and the third and fourth terms of the expressions indicate the size of the major counter. As an example, when $d = 3$ and $b = 8$, the memory size is 74,816 bits without the split counter and 46,200 bits with the split counter, which shows a large reduction in the amount of memory.

We should point out that overflows of the minor counters frequently occur, since each minor counter is given with a small bit length. When such an overflow occurs, the corresponding major counter is incremented

[17] It was mentioned that the SIT hardware implemented in [Gue16a] worked with 3.2GHz frequency.

Table 5: Number of clocks when minor counter in middle nodes overflows.

b	SIT	Incr. SIT	ELM
8	32	26	29
16	96	57	61
32	320	167	141
64	1,152	579	397
128	4,352	2,177	1,293

and all the minor counters associated with it are reset to zero. In ELM, b counters are used as the input for the plaintext part of tag generation by the MAC algorithm, that is, $b \times n/2$ bits should be verified by MAC. The usage of the split counter reduces the amount of counters stored in off-chips and the average latency of MACs because the input to the MAC algorithm is reduced[18].

More precisely, let $\mathtt{ctr}$ be the counter of the parent node and let $\mathtt{ctr}'_1, \mathtt{ctr}'_2, \ldots \mathtt{ctr}'_b$ be the counters of children nodes without the split counter, where b is the number of branches in the tree structure. In ELM, a tag T is generated as

$$T = \mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K,K'}\Big((\mathsf{ADD} \parallel \mathtt{ctr}), (\mathtt{ctr}'_1 \parallel \mathtt{ctr}'_2 \parallel \cdots \parallel \mathtt{ctr}'_b)\Big),$$

where $\mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K,K'}(N, M)$ calculates a tag from a nonce N and a plaintext M. Each counter is given by $n/2$ bits and $b \times n/2$ bits should be encrypted.

In contrast, we consider the tag generation when utilizing the split counter. Let $\mathtt{Mctr}$ and $\mathtt{mctr}$ be the major and minor counters of a parent node, respectively. Let $\mathtt{Mctr}'$ and $\mathtt{mctr}'_1, \mathtt{mctr}'_2, \ldots \mathtt{mctr}'_b$ be the major counter and minor counters of children nodes, respectively. In this case, unless the overflow of minor counters occurs, a tag is generated as

$$T = \mathsf{PXOR\text{-}MAC}.\mathcal{T}_{K,K'}\Big((\mathsf{ADD} \parallel \mathtt{Mctr} \parallel \mathtt{mctr}), (\mathtt{Mctr}' \parallel \mathtt{mctr}'_1 \parallel \cdots \parallel \mathtt{mctr}'_b)\Big).$$

Here, let s and t be the bit lengths of major and minor counters, respectively ($s + t = n/2$). When the split counter is used, the input is given with $s + bt$ bits. Since $s + bt \leq b(s + t)/2$, the input bit length of the MAC algorithm is reduced thanks to the split counter. In addition to the tag generation algorithm (*i.e.*, $\mathsf{PXOR\text{-}MAC}.\mathcal{T}$), Verify (*i.e.*, $\mathsf{PXOR\text{-}MAC}.\mathcal{V}$) and Update (*i.e.*, $\mathsf{PXOR\text{-}MAC}.\mathcal{VU}$) algorithms are performed with the split counter as well. The tags of leaf (or lowest) nodes can also be generated, verified, and updated in a similar manner even when the split counter is applied.

On the other hand, as described above, when the major counter is incremented due to the overflow of a minor counter, all minor counters associated with the major counter are reset to 0. Accordingly, we need to update all the tags where the nonces are reset. While the tag update of a tree without the split counter should update only d tags at each layer, the tag update with the split counter requires b times tag updates at the layer where a major counter is incremented (*i.e.*, the overflow of minor counter occurs), which is non-trivial in the whole tree update process (*i.e.*, Update).

Tables 5 and 6 show the numbers of clock cycles when an overflow occurs in MAC and AE, respectively. We evaluate the cases of five different numbers of branches. Since the reset counter is always the same value, the encryption result of reset counter in MAC can be pre-computed for both SIT and our trees. Hence, only the major counter (*i.e.*, $\mathtt{Mctr}$’) and nonce (*i.e.*, $(\mathsf{add} \parallel \mathtt{Mctr} \parallel \mathtt{mctr})$) should be computed if MAC offers the incremental property. Thus, ELM maintains superiority to SIT under the condition where an overflow occurs.

[18] Since only one major counter and b minor counters need to be verified, the number of clock cycles required per MAC is smaller than that without the split counter. However, when a major counter is incremented, it is necessary to recalculate all tags of the children nodes due to the reset of minor counters. In this case, the latency is greater than that without the split counter because this node requires b tag updates.

Table 6: Number of clocks when minor counter in leaf nodes overflows.

b	ℓ	SIT	ELM1	ELM2
8	512	136	59	49
	1,024	260	91	81
	2,048	512	155	147
	4,096	1,024	283	273
	8,192	2,048	539	529
16	512	136	59	49
	1,024	516	163	145
	2,048	1,024	291	275
	4,096	2,048	547	529
	8,192	4,096	1,059	1,041
32	512	520	179	145
	1,024	1,028	307	273
	2,048	2,048	563	531
	4,096	4,096	1,075	1,041
	8,192	8,192	2,099	2,065
64	512	1,032	339	273
	1,024	2,052	595	529
	2,048	4,096	1,107	1,043
	4,096	8,192	2,131	2,065
	8,192	16,384	4,179	4,113
128	512	2,056	659	529
	1,024	4,100	1,171	1,041
	2,048	8,192	2,195	2,067
	4,096	16,384	4,234	4,113
	8,192	32,768	8,339	8,209

We also found that the proposed AE is advantageous even with the split counter thanks to the simultaneous execution of pre-verification and update. In particular, when the number of branches increases, the proposed scheme becomes more advantageous in comparison with that without the split counter. The split counter is basically applied to trees covering a large memory size, where the amount of counters can be critical, and therefore we can confirm again the advantage of the proposed scheme.

8 Conclusion

We have presented ELM, a new memory encryption scheme with tree-based authentication. Unlike many recent proposals from computer architecture perspective, we focus on the internal MAC and AE modes, including their interactions, to reduce the entire latency of tree operations. ELM combines fully parallelizable MAC and AE modes and utilizes the incremental property of the MAC mode. Our AE mode is similar to OCB, however has a better decryption latency and it can be of independent interest as a stand-alone AE mode. We provide provable security results for these components as well as the whole authentication tree. Since Intel SGX's scheme (SIT) is a representative work on the same direction, we instantiated ELM using the same AES and compared ELM with SIT, and presented preliminary hardware implementations. The results showed that ELM achieves significantly lower latency, while keeping the comparable implementation size of SIT. Several future directions can be considered, as follows:

Other Instantiations. The use of AES is not a ultimate choice for latency. As we described, a low-latency block cipher or a tweakable block cipher (*e.g.*, PRINCE [BCG+12], QARMA [Ava17], and Midori [BBI+15]) will significantly improve our hardware results for both latency and size. It is even possible to consider using multiple primitives of possibly different block sizes for optimized performance. It is also interesting to study instantiations based on cryptographic permutations, *e.g.*, [NIS15, DEMS16, BKL+17].

Side-Channel Attacks. Cryptographic hardware frequently needs to be resistant against side-channel attacks for the application to memory encryption. It would be conducted in the future to design and evaluate side-channel-resistant hardware architecture for ELM. Here, the proposed architectures can employ any other block ciphers (satisfying the security criterion) and any type of architecture, instead of unrolled and pipelined AES encryption and decryption cores used in this paper. This indicates that we can easily realize a side-channel-resistant ELM hardware by replacing the AES cores with side-channel-resistant one because typical attackers attempt to retrieve the secret key of the underlying block cipher to break the confidentiality and authenticity.

For example, the masked round-based AES implementation in [SBHM20] achieves a far lower latency than any other conventional implementations based on a functional decomposition and byte-serial architecture, which would suit to the context of memory encryption. However, such masked implementations require a considerably large area overhead and on-the-fly random number generation (except for [Sug19, WM18]), which makes it impractical to unroll and pipeline the masked AES datapaths for high throughput. The usage of a (first-order) masking-friendly lightweight (tweakable) block cipher such as PRESENT [BKL+07], GIFT [BPP+17], and Skinny [BJK+16] would be a practical alternative to realize the side-channel resistance with a less area overhead and no on-the-fly randomness.

Furthermore, it would be interesting to design leakage-resilient TBC/permutation-based AE (*e.g.*, [DEM+17, BGP+19]) and MAC that enable low-latency operation and are suitable to be used with ELM.

References

ABB+19. Roberto Avanzi, Subhadeep Banik, Andrey Bogdanov, Orr Dunkelman, Senyang Huang, and Francesco Regazzoni. Qameleon v1.0. A Submission to the NIST Lightweight Cryptography Standardization Process, 2019.

Ava17. Roberto Avanzi. The QARMA block cipher family. *IACR Trans. Symm. Cryptol.*, 2017(1):4–44, 2017.

BBI+15. Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 411–436. Springer, Heidelberg, November / December 2015.

BCG+12. Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 208–225. Springer, Heidelberg, December 2012.

BDJR97. Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403. IEEE Computer Society Press, October 1997.

BGG94. Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 216–233. Springer, Heidelberg, August 1994.

BGP+19. Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. TEDT: a leakage-resistant AEAD mode. *IACR TCHES*, 2020(1):256–320, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/8400`.

BGR95. Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 15–28. Springer, Heidelberg, August 1995.

BJK+16. Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, Heidelberg, August 2016.

BKL$^+$07. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, Heidelberg, September 2007.

BKL$^+$17. Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 299–320. Springer, Heidelberg, September 2017.

BM97. Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Heidelberg, May 1997.

BN00. Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, Heidelberg, December 2000.

BPP$^+$17. Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 321–345. Springer, Heidelberg, September 2017.

BR02. John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 384–397. Springer, Heidelberg, April / May 2002.

DEM$^+$17. Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. ISAP – towards side-channel secure authenticated encryption. *IACR Trans. Symm. Cryptol.*, 2017(1):80–105, 2017.

DEMS16. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to Round 3 of the CAESAR competition, 2016.

Dwo07. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007. Available at `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf`.

Dwo10. Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. Standard, National Institute of Standards and Technology., 2010.

ECL$^+$07. Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. TEC-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 289–302. Springer, Heidelberg, September 2007.

Gue16a. Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. `http://eprint.iacr.org/2016/204`.

Gue16b. Shay Gueron. Memory Encryption for General-Purpose Processors. *IEEE Secur. Priv.*, 14(6):54–62, 2016.

HJ02. William Eric Hall and Charanjit S. Jutla. Parallelizable Authentication Trees. *IACR Cryptol. ePrint Arch.*, 2002:190, 2002.

HJ06. W. Eric Hall and Charanjit S. Jutla. Parallelizable authentication trees. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 95–109. Springer, Heidelberg, August 2006.

HSH$^+$08. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *USENIX Security 2008*, pages 45–60. USENIX Association, July / August 2008.

IK03. Tetsu Iwata and Kaoru Kurosawa. OMAC: One-key CBC MAC. In Thomas Johansson, editor, *FSE 2003*, volume 2887 of *LNCS*, pages 129–153. Springer, Heidelberg, February 2003.

KR11. Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *FSE 2011*, volume 6733 of *LNCS*, pages 306–327. Springer, Heidelberg, February 2011.

Kra01. Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, August 2001.

KS07. Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for two-round Advanced Encryption Standard. *IET Information Security*, 1(2):53–57, 2007.

LRW02. Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 31–46. Springer, Heidelberg, August 2002.

Mer88. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988.

Min07. Kazuhiko Minematsu. Improved security analysis of XEX and LRW modes. In Eli Biham and Amr M. Youssef, editors, *SAC 2006*, volume 4356 of *LNCS*, pages 96–113. Springer, Heidelberg, August 2007.

MM09. Kazuhiko Minematsu and Toshiyasu Matsushima. Generalization and extension of xex* mode. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 92-A(2):517–524, 2009.

MT06. Kazuhiko Minematsu and Yukiyasu Tsunoo. Provably secure MACs from differentially-uniform permutations and AES-based implementations. In Matthew J. B. Robshaw, editor, *FSE 2006*, volume 4047 of *LNCS*, pages 226–241. Springer, Heidelberg, March 2006.

NIS15. NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 2-2, Federal Information Processing Standards Publication, 2015.

NIS19. NIST. Lightweight Cryptography Project. `https://csrc.nist.gov/Projects/Lightweight-Cryptography`, 2019.

NRS14. Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. Reconsidering generic composition. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 257–274. Springer, Heidelberg, May 2014.

RBBK01. Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 196–205. ACM Press, November 2001.

RCPS07. Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *MICRO*, pages 183–196. IEEE Computer Society, 2007.

Rog02. Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, November 2002.

Rog04. Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 16–31. Springer, Heidelberg, December 2004.

SBHM20. Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-latency hardware masking with application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):300–326, 2020.

SNR$^+$18. Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, José A. Joao, and Moinuddin K. Qureshi. Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *MICRO*, pages 416–427. IEEE Computer Society, 2018.

Sug19. Takeshi Sugawara. 3-share threshold implementation of AES s-box without fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):123–145, 2019.

TSB18. Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *ASPLOS*, pages 665–678. ACM, 2018.

UMHA16. Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths - toward efficient cbc-mode implementation. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 538–558. Springer, 2016.

UMM$^+$20. Rei Ueno, Sumio Morioka, Noriyuki Miura, Kohei Matsuda, Makoto Nagata, Shivam Bhasin, Yves Mathieu, Tarik Graba, Jean-Luc Danger, and Naofumi Homma. High throughput/gate AES hardware architectures based on datapath compression. *IEEE Trans. Computers*, 69(4):534–548, 2020.

UWM19. Thomas Unterluggauer, Mario Werner, and Stefan Mangard. MEAS: memory encryption and authentication secure against side-channel attacks. *Journal of Cryptographic Engineering*, 9(2):137–158, June 2019.

WM18. Felix Wegener and Amir Moradi. A first-order SCA resistant AES without fresh randomness. In *COSADE*, volume 10815 of *Lecture Notes in Computer Science*, pages 245–262. Springer, 2018.

YEP$^+$06. Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *ISCA*, pages 179–190. IEEE Computer Society, 2006.