On Protecting Integrity and Confidentiality of Cryptographic File System for Outsourced Storage

Aaram Yun
University of Minnesota
Minneapolis, MN 55455
aaram@cs.umn.edu

Chunhui Shi
University of Minnesota
Minneapolis, MN 55455
cshi@cs.umn.edu

Yongdae Kim
University of Minnesota
Minneapolis, MN 55455
kyd@cs.umn.edu

ABSTRACT

A cryptographic network file system has to guarantee confidentiality and integrity of its files, and also it has to support random access. For this purpose, existing designs mainly rely on (often ad-hoc) combination of Merkle hash tree with a block cipher mode of encryption. In this paper, we propose a new design based on a MAC tree construction which uses a universal-hash based stateful MAC. This new design enables standard model security proof and also better performance compared with Merkle hash tree. We formally define the security notions for file encryption and prove that our scheme provides both confidentiality and integrity. We implement our scheme in coreFS, a user-level network file system, and evaluate the performance in comparison with the standard design. Experimental results confirm that our construction provides integrity protection at a smaller cost.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms

Algorithms, security

Keywords

File encryption scheme, cryptographic file system, MAC tree, Merkle hash tree, universal-hash based MAC, provable security

1. INTRODUCTION

1.1 Background

Cryptographic file system is a convenient solution for protecting data at rest. It offers automatic and transparent encryption, and in contrast to block-device level encryption, it provides more flexible key and user management. Such a system has to guarantee both confidentiality and integrity of its files, that is, it should give protection against information leakage, and also against unauthorized alteration of data.

Although it is possible to use cryptographic file systems for encrypting data locally, more care is needed if one wants to build a cryptographic network file system on top of outsourced cloud storage services. For example, a cryptographic file system on a laptop may need only protection from one-time data loss (theft, or missing laptop), but when the encrypted data is stored in a third-party storage such as Amazon S3 [2], the attacker may potentially observe many (encrypted) modifications to the stored data and also be able to adaptively modify the data.

Therefore, we consider the scenario where the attacker has adversarial control over the data storage. For example, a malicious storage provider might delete an update of a file and instead 'roll-back' the file's content to an outdated previous version.

A conventional solution to protect against such cases is to use a block cipher mode of encryption for confidentiality, and combine that with a Merkle hash tree construction for integrity; a file is divided into small blocks, and the tree of hash values of the blocks is built, and the root of the tree is authenticated. This provides logarithmic-time random access of file content. Figure 1 illustrates a ternary Merkle hash tree.

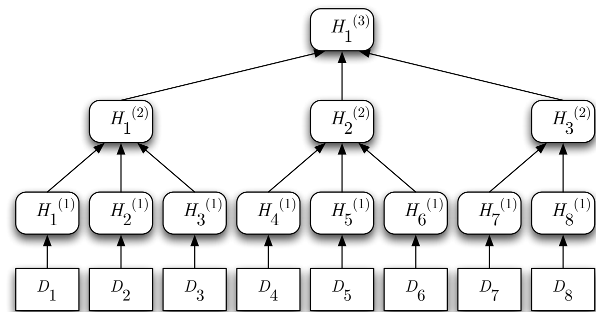

Figure 1: Merkle hash tree construction

1.2 Objectives

In this paper, we have two goals in mind:

1. Security: to provide a provably secure design of a cryp-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCSW'09, November 13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-784-4/09/11 ...$10.00.

tographic file system along with rigorous security definition

2. Efficiency: to optimize performance in comparison with previous solutions.

We want our construction to provide provable security of confidentiality and integrity. We define the notions of security, and prove that our scheme meets the security goals. We firmly believe that security proofs are important: not because unproven schemes necessarily have critical flaws, but because rigorous definition and proof *ensure* us what security do we have. Even such an 'obviously secure' construction like Merkle hash tree can be sometimes misused if not careful: for example, some proposed schemes build the Merkle tree out of the plaintext of the file content, and protect the content by storing the encrypted content. This leaks information about the file content, for the simple reason that the Merkle tree is a deterministic function of its input: by looking at the Merkle tree, the attacker may compare various versions of a file and see whether some of the versions are identical in certain blocks or not, i.e., such a scheme is not semantically secure.

For efficiency, our goal is to obtain much smaller computational overhead of cryptographic operations in comparison with Merkle hash tree based solutions, at the same time with comparable or better communication overhead. In cryptographic network file systems, typically communication is the bottleneck and computational overhead for cryptographic operations takes shorter time in comparison. But saving the computational overhead for cryptography is still meaningful for practical systems, for the saved computational cycles can contribute to responsiveness of the whole system or can be used for computations other than cryptographic operations.

Again, with encryption, there is a natural limit for optimization given a fixed amount of data to be processed, unless parallelization or pipelining via multi-core or GPU is used.

However, authentication of data can be optimized quite a bit, especially in comparison with Merkle hash tree. Also, computational cost for authentication is typically smaller than that of encryption, but not too much smaller. For example, if we look at the benchmark data made available by eBACS [5], except few unusual cases, in most environments SHA-1 is at best only about 1.5–1.6 times faster than AES-128 as a rule of thumb, and often slower than that. And SHA-256 is typically slower than AES-128. This means that, in order to build a Merkle hash tree to protect integrity, one has to spend comparable, or sometimes more computational resources than encryption. Judicious choices of the shape, serialization, and caching of the tree would help reducing the cost, but at the minimum, each block has to be hashed and that cannot be reduced any further.

1.3 Universal-hash based MAC tree

Instead, we propose using a universal-hash based stateful MAC [8, 23]. Although in our implementation we use Poly1305-AES [3], in general any universal-hash based MAC can be used. A universal hash function is a keyed, non-cryptographic hash function which guarantees only collision resistance or related properties. Since it does not mix and scramble the input as thoroughly as a cryptographic hash function does, usually it is much simpler and faster. In order to be used as a cryptographic data authenticator, a nonce is encrypted by a block cipher and that output is XORed to the output of the universal hash function. Because the costly block cipher operation is used only once, irrespective of the data size, a universal-hash based MAC performs better with longer amount of data, relatively. This performance characteristic fits well with integrity protection for storage security: we propose a new n-ary MAC tree construction out of a universal-hash based MAC algorithm, with n much larger than 2 so that all children of a node fit in a disk block. Also, in contrast to a hash tree or a stateless MAC tree, where lower authentication tags themselves should be authenticated in a higher node, we need only to authenticate *counters* in the higher nodes, not the authentication tags. Since we may use a counter much smaller than a secure authentication tag (in our case, we use 4-byte counters and 16-byte tags), this gives further performance enhancement.

We implement our scheme on coreFS [9], a user-level network file system, and compare the performance of our scheme with that of a Merkle hash tree based scheme. The experimental results confirm that our construction provides integrity protection at a smaller cost than the case of the Merkle hash tree.

The following is the organization of this paper. In Section 2, we discuss related work. In Section 3, we give definition of file encryption schemes, and also define the notions of security. In Section 4, we present our file encryption scheme. In Section 5, we give implementation details, and in Section 6, we show the performance analysis. In Section 7, we conclude the paper. Additionally, we sketch the security proofs in Appendix A.

2. RELATED WORK

There are many examples of cryptographic file systems that use the Merkle hash tree or MAC tree (built from stateless MAC and similar to the Merkle tree) to protect integrity; a few examples include [1], [10], [11], [14], [16], [17], [20].

Oprea and Reiter [19] propose three file encryption schemes for cryptographic file systems. Their main goal is to reduce the amount of extra space for integrity protection, and for this they suggest schemes which take advantage of non-randomness of some file blocks: in one scheme, they use a wide-block cipher and explicitly protect the integrity only for blocks with random contents, and in another scheme, they compress non-random file blocks and store the authentication tag in the space saved by the compression. We point out that their schemes are not semantically secure: as a default authentication mechanism they use the Merkle tree with plaintext blocks, and this leaks information as explained in the introduction, and also because they treat random blocks and non-random blocks differently, this leaks yet another information about the file blocks, that is, whether they are random or not.

Gjøsteen [13] defines and compares various notions of security for disk encryption. His work is for disk volume encryption schemes, but many of the definitions are also applicable to file encryption schemes as well.

We assume the existence of a fixed-size per file, public, trusted storage. This is necessary for protection from 'rollback' of content to previous versions. Even if each party who has legitimate access to the file maintains a private state for the file, still the storage provider may 'hide' someone's update to the file from others, while showing consistent view to each individual. Mazières and Shasha defined the no-

tion of 'fork consistency' [18], which is weaker, but probably the strongest possible guarantee of integrity when there's no public trusted storage. Li, Krohn, Mazières and Shasha developed SUNDR file system [16] which satisfies fork consistency.

Carter and Wegman proposed the notion of universal hashing in [8]. Universal hashing has numerous uses throughout computer science. The application of universal hashing to message authentication was suggested first by Wegman and Carter [23]. Originally they suggested using one-time pad to produce the authentication tag, but soon Brassard suggested using a block cipher and a nonce [7]. There are many examples of MAC algorithms following this paradigm, and, in fact, some of the fastest authentication algorithms are of this type. Some modern examples include Poly1305-AES [3], UMAC, and VMAC [15].

3. PRELIMINARIES

3.1 File encryption schemes

In order to provide efficient random access of data, we cannot simply treat the content of a file as a single, consecutive data and encrypt and authenticate it accordingly. Therefore, we model a file Φ to store and maintain its contents in 'chunks'. Let $D = D_1 \cdots D_n$ be the content of Φ, where $D_i \in \{0,1\}^d$ for some d. We call D_i the ith *file block* of the file Φ. For simplicity, we do not consider files with data smaller than a file block: this can be handled by fixing some padding scheme, and all of our results can be applied to such cases. From now on, we call n the *length* of the file content $D_1 \cdots D_n$.

We also use a block cipher for our construction. So, we have two kinds of 'blocks'. In case of possible confusion, we'll use the term 'file block' to distinguish the former from the latter. Let b be the block size of the block cipher in bits. Concretely, we may choose $d = 32768$, which is 4 KB, and we may choose AES as our block cipher, and this will fix $b = 128$.

We model a file Φ as a quintuple of algorithms (Read, Length, Update, Delete, Append). All of these operations implicitly involve a symmetric key, called the *file encryption key* of the file. The problem of generation and distribution of the file encryption key is out of the scope of this paper: we simply assume that there is a secure mechanism for distributing the file encryption key to anyone who are authorized to access the file.

We assume that the (encrypted) file content will be stored in an untrusted storage, which we denote by $\mathcal{S}$. In addition to that, we assume that there is a separate, fixed-size, public, trusted storage space $\mathcal{T}$ for each file. Therefore, the state of a file is completely described by $(t, s) \in \mathcal{T} \times \mathcal{S}$, and, in addition to explicit input arguments, each file operation interacts with the state information of the storage spaces, and may read and update the state (t, s). Each algorithm can output $\perp$, indicating failure, and halt. In that case, if the state at the beginning of the algorithm was $(t, s) \in \mathcal{T} \times \mathcal{S}$, then at the end of failure, it is $(t, s') \in \mathcal{T} \times \mathcal{S}$, where t is the same as at the beginning, but s' is some state which may or may not be same as s. We also assume that there is a fixed state (t_0, s_0), and the state of Φ is initialized to (t_0, s_0) when the file is initialized as an empty file.

The input and output format of each algorithm is as follows:

- $\mathsf{Read}(k)$, for $k = 1, \ldots, 2^L$ (where L is some fixed large number which depends on Φ), either returns some $D \in \{0,1\}^d$, or $\perp$.
- $\mathsf{Length}()$ either returns an integer $n \geq 0$, or $\perp$.
- $\mathsf{Update}(k, D)$, for $k = 1, \ldots, 2^L$ and $D \in \{0,1\}^d$, either returns $\top$, indicating success, or $\perp$.
- $\mathsf{Append}(D)$, for $D \in \{0,1\}^d$, either returns $\top$, indicating success, or $\perp$.
- $\mathsf{Delete}()$ either returns $\top$, indicating success, or $\perp$.

3.2 File contents and soundness

Actually, the content of the file has to be defined operationally from the file operation requests. Let $(q_1, \ldots, q_r)$ be a sequence of file operation requests. We define the content $\mathcal{C}(q_1, \ldots, q_r)$ of the file with respect to the sequence recursively as follows:

- $\mathcal{C}() \stackrel{\text{def}}{=} \epsilon$; the content for null sequence of operations is defined as the empty data.
- If $\mathcal{C}(q_1, \ldots, q_{r-1}) = D_1 \cdots D_n$, where $D_i \in \{0,1\}^d$, then $\mathcal{C}(q_1, \ldots, q_r)$ is defined by
  - Read, Length, and any failed requests do not modify the content.
  - $\mathcal{C}(q_1, \ldots, q_r) \stackrel{\text{def}}{=} D_1 \cdots D_{i-1} \hat{D} D_{i+1} \cdots D_n$, if $q_r = \mathsf{Update}(i, \hat{D})$ and $i \leq n$ and the request q_r is successful.
  - $\mathcal{C}(q_1, \ldots, q_r) \stackrel{\text{def}}{=} D_1 \cdots D_n \hat{D}$, if $q_r = \mathsf{Append}(\hat{D})$, and the request q_r is successful.
  - $\mathcal{C}(q_1, \ldots, q_r) \stackrel{\text{def}}{=} D_1 \cdots D_{n-1}$, if $q_r = \mathsf{Delete}()$ and $n \geq 1$, and the request q_r is successful.

Intuitively, this means that the content of a file at any time is defined by previous sequence of successful file operation requests.

We require that the file Φ is *sound*, that is, under no alteration of $\mathcal{S}$ by an attacker, Φ faithfully stores and maintains its content. That is, if $(q_1, \ldots, q_r)$ was the sequence of requests made so far, and if $\mathcal{C}(q_1, \ldots, q_r) = D_1 \cdots D_n$, then, after that,

- If $\mathsf{Read}(i)$ request is made, then it will return D_i if $1 \leq i \leq n$, and it will return $\perp$ otherwise.
- If $\mathsf{Length}()$ request is made, then it will return n.
- If $\mathsf{Update}(i, \hat{D})$ request is made, then it will fail if and only if $i > n$.
- If $\mathsf{Append}(\hat{D})$ request is made, then it will succeed.
- If $\mathsf{Delete}()$ request is made, then it will fail if and only if $n = 0$.

3.3 Security goals

Integrity.

We define the integrity of a file as infeasibility of alteration of the file content, under chosen message attack. More concretely, after Φ is initialized, we let an attacker A to freely interact with Φ, making any file operation requests, and we also allow the attacker to feed arbitrary state information $s' \in \mathcal{S}$, which may be different from the state output of the last successful request.

We say that the attacker A succeeds violating the integrity of Φ, if A ever makes a successful read request $\mathsf{Read}(i)$, and obtain $\hat{D} \neq D_i$, where D_i is the ith data block of the content of Φ when A makes that Read request.

(Technically, we may consider that the integrity is violated also when the attacker may alter the length of the file content. But, in our construction, we'll store and update the length information in the trusted storage, therefore we do not have to consider this case.)

Let $\mathrm{Adv}_\Phi^{\mathrm{int}}(A)$ be the probability that A may violate the integrity of Φ. This probability is over the internal coin tosses of A, coin tosses of Φ (although in our construction, Φ consists of deterministic algorithms), and over the randomness of key materials used by Φ.

Informally, we say that Φ provides secure integrity, if $\mathrm{Adv}_\Phi^{\mathrm{int}}(A)$ is negligible for all efficient attackers A.

Confidentiality.

We define the confidentiality of a file as infeasibility of an attacker to learn any information about any file block, other than by explicitly reading the file block; even if the attacker can eavesdrop or coerce a party with legitimate access of the file to read some portions of the file, at least other, unread portions are safe.

In order to define the confidentiality, let's define 'left-or-right' oracle by $\mathrm{LR}(b, M_0, M_1) \stackrel{\mathrm{def}}{=} M_b$, for $b = 0$ or 1. For the definition of integrity, assume that at the initializition of the file, b is randomly chosen from {0,1}, and assume that the attacker can make the same type of file operation requests, except that, now the format of Update and Append requests are altered: the attacker requests $\mathsf{Update}(i, \hat{D}_0, \hat{D}_1)$, and this is translated as $\mathsf{Update}(i, \mathrm{LR}(b, \hat{D}_0, \hat{D}_1))$ and given to the file encryption scheme. Similarly, the attacker's request $\mathsf{Append}(\hat{D}_0, \hat{D}_1)$ is translated as $\mathsf{Append}(\mathrm{LR}(b, \hat{D}_0, \hat{D}_1))$. (Because $\mathsf{Update}(i, \hat{D}, \hat{D}) = \mathsf{Update}(i, \hat{D})$, this new game is generalization of the previous.)

With respect to these new types of queries, we can similarly define the content $\mathcal{C}(q_1, \ldots, q_r)$ as $(D_1 \cdots D_n, D'_1 \cdots D'_n)$, where $D_1 \cdots D_n$ is the 'left' content, i.e., the content should be $D_1 \cdots D_n$, if $b = 0$, and $D'_1 \cdots D'_n$ is the 'right' content. We can use similar recursive definition as before: for example,

$$\begin{aligned}\mathcal{C}(q_1, \ldots, q_r) \stackrel{\mathrm{def}}{=} (&D_1 \cdots D_{i-1} \hat{D} D_{i+1} \cdots D_n,\\ &D'_1 \cdots D'_{i-1} \hat{D}' D'_{i+1} \cdots D'_n),\end{aligned}$$

if $\mathcal{C}(q_1, \ldots, q_{r-1}) = (D_1 \cdots D_n, D'_1 \cdots D'_n)$,
$q_r = \mathsf{Update}(i, \hat{D}, \hat{D}')$, $i \leq n$, and the request q_r is successful.

We reject any attacker A, if he ever makes a query $\mathsf{Read}(i)$, when the content is $\mathcal{C} = (D_1 \cdots D_n, D'_1 \cdots D'_n)$ with $D_i \neq D'_i$. At the end of the session with the file Φ, the attacker A outputs 0 or 1. Let p be the probability that the output of A is equal to b, which was chosen at the initialization of the file. Let's define the advantage of A as $\mathrm{Adv}_\Phi^{\mathrm{conf}}(A) \stackrel{\mathrm{def}}{=} |p - 1/2|$.

Informally, we say that Φ provides secure confidentiality, if $\mathrm{Adv}_\Phi^{\mathrm{conf}}(A)$ is negligible for all efficient attackers A.

3.4 Universal-hash based MAC schemes

The main component of our file encryption scheme is a universal-hash based MAC.

An ϵ-almost XOR-universal (AXU for short) hash function is a keyed function family $H_K : \mathcal{D} \to \{0,1\}^r$ which satisfies the following: for any distinct X, $Y \in \mathcal{D}$ and any $C \in \{0,1\}^r$, the probability that $H_K(X) \oplus H_K(Y) = C$ is at most ϵ with respect to randomly chosen K. There are many such constructions, and one canonical example is 'polynomial hashing': to compute $H_K(M)$, interpret the message M as coefficients of a polynomial, and evaluate the polynomial at the value K given as the random key. This is a very lightweight construction, and some of modern universal hash functions are more than 10 times faster than most cryptographic hash functions or block ciphers, for example.

But an AXU hash function is not a cryptographically secure MAC, and it cannot be directly used for data authentication; once an attacker sees a few message-hash pairs $(M, H_K(M))$, it is easy to forge a new message-hash pair $(M^*, H_K(M^*))$, even when the attacker has no knowledge of K.

One way of using an AXU hash function for the purpose of authentication is to 'blind' the hash values by pseudorandom numbers produced by a block cipher. Let $E_K(\cdot)$ be a block cipher, for example AES. Pick keys K and K' randomly, for E and h, respectively. In order to authenticate a message M, then we pick a nonce N, and compute the authentication tag T as

$$T \leftarrow \mathcal{M}_{K,K'}(N, M) \stackrel{\mathrm{def}}{=} E_K(N) \oplus H_{K'}(M).$$

When verifying the authenticity of (M, N, T), the verifier again checks if $T \stackrel{?}{=} \mathcal{M}_{K,K'}(N, M)$ holds.

Note that, however large M is, only one block cipher operation is used during the calculation of $\mathcal{M}(N, M)$, while the universal hashing is very lightweight. This is the main reason for the extreme efficiency of this type of constructions.

It is proven [22, 4] that, if the block cipher E is secure as a pseudorandom function, then, as long as no nonce is re-used during the generation of tags, forging a new valid (M, N, T) tuple is infeasible, even after the attacker has seen many such tuples before, either by eavesdropping or by active manipulation of tag generation. Also, nonce repetition is allowed for the verification: the scheme is secure even when an attacker tries many different M and T with one N.

4. DESIGN

4.1 A new MAC tree construction

Our file encryption scheme is based on a new MAC tree construction, using a stateful, universal-hash based MAC scheme described in the previous section. In Figure 2, we illustrate one such 3-ary MAC tree, where $\mathcal{M}$ denotes the MAC algorithm used and a dashed line denotes the nonce used in the authentication. Let a be the arity of the tree. The $N_i^{(j)}$ are all counters, and a of the jth level counters $N_{(i-1)a+1}^{(j)}, \ldots, N_{ia}^{(j)}$ are concatenated and authenticated us-

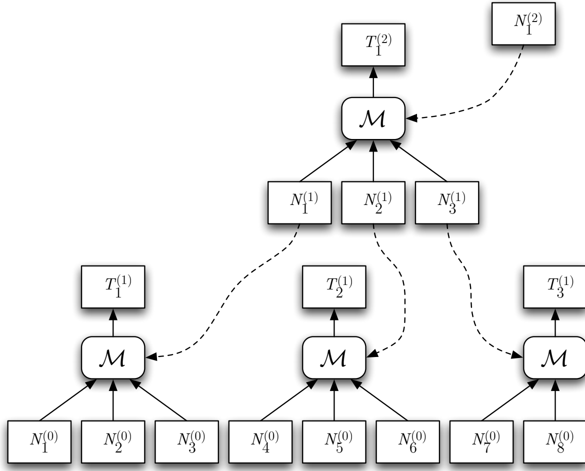

Figure 2: MAC tree construction

ing the $(j+1)$th level parent counter $N_i^{(j+1)}$ to produce the authentication tag $T_i^{(j+1)}$:

$$T_i^{(j+1)} \leftarrow \mathcal{M}_K(N_i^{(j+1)}, N_{(i-1)a+1}^{(j)} \| \cdots \| N_{ia}^{(j)}).$$

The 'root counter' $N_1^{(l)}$, where l is the depth of the tree, is stored in a trusted storage, along with the depth l and the total number m of leaf counters. The rest of the tree, both $N_i^{(j)}$ and $T_i^{(j)}$, can be stored in an untrusted storage.

When we need to increase one of the leaf counters $N_i^{(0)}$, we first follow the path leading up to the root counter, and fetch all the relevant counters (the ancestors of $N_i^{(0)}$ and their direct siblings) and tags to check that the counters and the tags 'match'. After that, we increase the leaf counter and its ancestor counters, re-compute all the affected authentication tags, and store the result accordingly: again the root counter is stored in the trusted storage, and the rest of the tree in the untrusted storage. Appending a new leaf counter can be handled similarly.

The goal of this MAC tree construction is, whenever all the verifications are passed, we may regard the leaf counters as if they are stored in a trusted storage. The main intuition is the transfer of trust from the root counter to the lower level counters; since we trust the root counter (stored in a trusted storage), when the tag verification for the root level is successful, we can trust the children counters of the root counter. Repeating this process, we pass the trust down to the leaf counters. Even though the underlying MAC scheme is secure only when the nonce is never repeated, in this way, we can show that the probability of nonce re-use during the MAC tree computation is negligible, if the underlying MAC scheme itself is secure.

In short, by this MAC tree construction, we can maintain arbitrary number of 'trusted' leaf counters, using a *fixed size* per file, public trusted storage, and an untrusted storage.

4.2 Authenticated encryption scheme for files

Now, using the MAC tree construction we can build a file encryption scheme. Let $D = D_1 \cdots D_n$ where $D_i \in \{0,1\}^d$ the content of the file. We apply a nonce-based authenticated encryption scheme to each file block D_i, using the leaf counter $N_i^{(0)}$:

$$(C_i, T_i) \leftarrow \mathcal{E}_K(N_i^{(0)}, D_i),$$

and store the ciphertext C_i and the authentication tag T_i to the untrusted storage, along with the MAC tree for maintaining the leaf counters $N_i^{(0)}$.

In fact, since we already need an efficient MAC scheme for the purpose of building the MAC tree, we can construct a simple authenticated encryption scheme as composition of the CTR encryption mode of operation and the MAC scheme. Figure 3 illustrates the composite scheme: using the counter $N_i^{(0)}$, we generate a pseudorandom sequence by the CTR mode, the file block D_i is XORed to this sequence to produce the ciphertext block C_i, and we again use the counter $N_i^{(0)}$ to authenticate C_i, producing the authentication tag T_i.

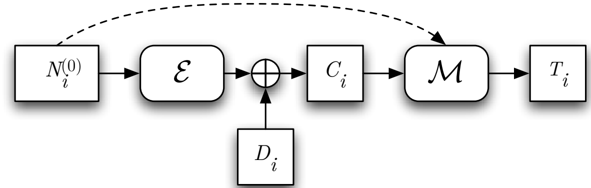

Figure 3: Blockwise authenticated encryption using leaf counters

Special care is needed to never re-use a nonce throughout our construction. Note that here the counter $N_i^{(0)}$ is used both for encryption and authentication. In fact, we do not use the counter directly, but use different encoding schemes to generate non-repeating nonces:

$$\begin{aligned} C_i &\leftarrow E_{K_e}(\langle i, 1, N_i^{(0)} \rangle_e) \cdots E_{K_e}(\langle i, d/b, N_i^{(0)} \rangle_e) \oplus D_i, \\ T_i &\leftarrow \mathcal{M}_{K_a,K_h}(\langle i, 0, N_i^{(0)} \rangle_a, C_i) \\ &\stackrel{\text{def}}{=} E_{K_a}(\langle i, 0, N_i^{(0)} \rangle_a) \oplus H_{K_h}(C_i), \end{aligned}$$

where the key $K = (K_e, K_a, K_h)$ consists of the block cipher key K_e for encryption, the block cipher key K_a for authentication, and the AXU hash function key K_h. And $\langle i, j, N \rangle_e$ is a nonce encoding scheme satisfying $\langle i, j, N \rangle_e = \langle i', j', N' \rangle_e$ iff $i = i'$, $j = j'$, and $N = N'$. Similarly, $\langle i, j, N \rangle_a$ is yet another nonce encoding scheme. It is also required that the two nonce encoding schemes never overlap: $\langle i, j, N \rangle_e \neq \langle i', j', N' \rangle_a$ for all i, j, N, i', j', N'. Actually, these nonce encodings should be used also in the MAC tree construction.

Then, the following is more detailed description of our file encryption scheme:

Let $\mathcal{T}$ be the trusted storage, and $\mathcal{S}$ be the untrusted storage. A file Φ stores and maintains its content $D = D_1 \cdots D_n$, where $D_i \in \{0,1\}^d$ for $i = 1, \ldots, n$. In order to support this, we maintain the leaf counters $N_1^{(0)}, \ldots, N_m^{(0)}$ for some $m \geq n$, along with the whole MAC tree construction consisting of higher level counters $N_i^{(j)}$ and tags $T_i^{(j)}$. The trusted storage $\mathcal{T}$ stores $(n, m, l, N_1^{(l)})$, where n is the length of the content D, $m \geq n$ is the length of the 'active' leaf counters, l the depth of the MAC tree, and $N_1^{(l)}$ the root counter. The untrusted storage $\mathcal{S}$ stores the rest of the MAC tree, and

the ciphertext blocks $C_1, \ldots, C_n$ corresponding to the file blocks $D_1, \ldots, D_n$, and also the file block tags T_i.

Initially, $\mathcal{T}$ is initialized to $(n, m, l, N_1^{(l)}) = (0, 0, \perp, \perp)$, and the MAC tree is initialized as a null tree: $N_i^{(j)} = \perp$ for all i, j.

A file Φ supports Read, Length, Update, Append, and Delete operation requests. They can be described as:

- Read(k): in order to read the kth file block D_k, we let $i_0 \stackrel{\text{def}}{=} k$, and define i_{j+1} as the index of the parent counter of i_j. Verify if

$$T_{i_j}^{(j)} \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle i_j, j, N_{i_j}^{(j)} \rangle_a, D_{i_j}^{(j)})$$

  for $j = 1, \ldots, l$. Here $D_{i_j}^{(j)}$ is the concatenation of children of the counter $N_{i_j}^{(j)}$. If this verification is successful, then further verify the ciphertext block C_k using the leaf counter $N_k^{(0)}$:

$$T_k \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle k, 0, N_k^{(0)} \rangle_a, C_k).$$

  If this verification is successful, then decrypt D_k using the ciphertext block C_k and the counter $N_k^{(0)}$, and return D_k.

- Length(): simply return the length n which was stored in the trusted storage $\mathcal{T}$.

- Update($k, \hat{D}$): Similar to the Read(k) request, verify the counters $N_{i_j}^{(j)}$ which are ancestors of the kth leaf counter $N_k^{(0)}$. When the verification is successful, update those counters by $N_{i_j}^{(j)} \leftarrow N_{i_j}^{(j)} + 1$, regenerate the affected tags

$$T_{i_j}^{(j)} \leftarrow \mathcal{M}_{K_a,K_h}(\langle i_j, j, N_{i_j}^{(j)} \rangle_a, D_{i_j}^{(j)})$$

  for $j = 1, \ldots, l$, encrypt the new kth file block $\hat{D}$ using $N_k^{(0)}$ to produce new ciphertext block C_k, authenticate C_k by generating a tag

$$T_k \leftarrow \mathcal{M}_{K_a,K_h}(\langle k, 0, N_k^{(0)} \rangle_a, C_i),$$

  and update both $\mathcal{T}$ and $\mathcal{S}$.

- Delete(): If $n = 0$, then output $\perp$ and halt. Otherwise, remove the nth ciphertext block and tag by $C_n \leftarrow \perp$, $T_n \leftarrow \perp$, and update n by $n \leftarrow n - 1$. Update $\mathcal{T}$ and $\mathcal{S}$ accordingly. Note that we *do not* reset or delete the nth leaf counter $N_n^{(0)}$, in order to avoid nonce re-use by later Append operations. This is where n and m may become unequal.

- Append($\hat{D}$): If $n < m$, then this is essentially the same as Update($n + 1, \hat{D}$). If $n = m$, then we have to append a new leaf counter $N_{n+1}^{(0)}$. Again we verify all the possible ancestor counters, and if the verification is successful, then initialize $N_{n+1}^{(0)} \leftarrow 1$ (along with other non-initialized ancestors), and increase the existing ancestors by 1, and follow steps similar to the Update request. At the end, we increase both n and m by 1, and store them in the trusted storage $\mathcal{T}$.

4.3 Security of the scheme

Intuitively, it is easy to see that, if we do not use the MAC tree construction, and instead manage all of the leaf counters $N_k^{(0)}$ in the trusted storage, then our scheme guarantees both confidentiality and integrity of the file content: all the block counters $N_k^{(0)}$ are properly incremented, and no nonce is repeated, and each file blocks are encrypted and authenticated independent from each other. Essentially, the confidentiality and the integrity of the resulting scheme comes from the security of the composite authenticated encryption scheme, and the nonce-respecting property (since we use trusted storage for nonces).

In our actual scheme, instead of managing the leaf counters in the trusted storage, we store them in our MAC tree and store only the root counter in the trusted storage. Still, from the way the MAC tree is constructed, our trust of the root counter can be transferred to its descendants, as long as all of the tag verifications along the path are successful. In this way, we can ensure that our MAC tree construction can safely replace counters stored in a trusted storage, except negligible probability of successful attack.

In the Appendix, we discuss the security proof of our scheme in more details.

4.4 Re-use of block cipher keys

Previously, we described our scheme as using a file encryption key of form $K = (K_e, K_a, K_h)$, where K_e is the block cipher key for encryption, K_a is the block cipher key for authentication, and K_h is the AXU hash function key. Actually, it is safe to use the same random bits for both K_e and K_a: the reason is that we use separate, non-overlapping nonce encodings $\langle\cdot\rangle_e$ and $\langle\cdot\rangle_a$ for encryption and authentication. The output of the block cipher E is used as pseudorandom sequence to 'blind' the plaintext blocks and the AXU hash values, and since no nonce is re-used, all these pseudorandom outputs are essentially independent from each other.

5. IMPLEMENTATION

We use coreFS [9] as our platform for implementing our prototype cryptographic file system. CoreFS is a FUSE [12] based network file system which provides core functionalities of distributed file system.

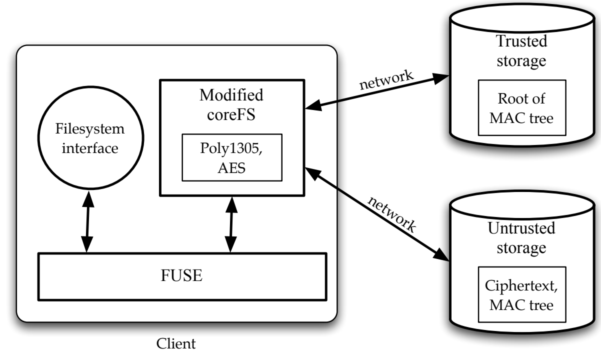

Figure 4: Architecture diagram

Fig. 4 represents basic architecture of our implementation. We modified coreFS to incorporate our scheme, using

Poly1305 and AES as cryptographic building blocks. File system operations of applications are relayed by FUSE to the modifed coreFS, and coreFS sends and receives encrypted and authenticated data to and from storages.

In our prototype implementation, actually we made some simplifications: we did not use a separate trusted storage and stored the state as an extended metadata of the file. Also, we stored the file encryption key as an extended metadata, because the distribution mechanism of file encryption key is out of scope of this paper.

Both for the Merkle hash tree and for our construction, the choices of the arity of tree, method of serialization, caching, and other practical implementational details could affect the performance. We choose to use a file block of 4 KB, and for algorithms, we use the AES block cipher and the Poly1305-AES MAC algorithm to build a 64-ary MAC tree. This makes our tree very flat, but we need only to authenticate the counters in the tree, the size of input which is to be processed together is only 256 bytes, and the speed of Poly1305 compensates this larger input size.

In our file system, the client file system interacts with the server by network. We store all the ciphertext blocks together in a file in the server, and store the rest of data, 'cryptographic metadata', in a separate file. Of course, the client file system processes two files together, and the application programs which use the client file system do not see two separate files, but only combined one.

In our prototype implementation, for simplicity we used unlimited amount of cache, that is, once a counter is read into memory and verified, then it will not be erased until the file is closed. Also, for write operation, change of the MAC tree data will be uploaded to the storage server only when the file is closed or flushed. It is true that choices of caching strategy and update schedule for cryptographic metadata affects the performance of our scheme and also that of Merkle hash tree based schemes, but in an actual system these choices should depend on system requirements. We compare the performance of our algorithm with composition schemes of AES-CTR and SHA-1 (truncated to 128 bits) based 16-ary Merkle hash tree, and we implement them with the same policy with respect to caching and metadata update.

6. PERFORMANCE EVALUATION

For performance evaluation, we used two machines, one for client and one for storage server. Both are DELL Optilex GX620 with Pentium 4 CPU of 3.0 GHz. Each machine has 1 GB RAM and 80 GB hard disk. They are connected locally by a 1 Gb ethernet.

Bonnie++ [6] is a hard disk and file system benchmark program. It does character and block read/write transactions to a single large file. We ran Bonnie++ benchmark to measure the performance of sequential read/write operations. We repeatedly tested this for 2 GB files, and, in order to break down the overall time into those for I/O, for encryption, and for authentication, we measured elapsed CPU cycles at various points during the benchmark. We compared the performance of our scheme with Merkle hash tree based constructions. Overall, network overhead was about 65–85%, in comparison with other costs. Fig. 5 shows the computational overhead of authentication and encryption/decryption for different tests (in 10^6 cycles). In case of Merkle hash tree based constructions, the cost for authentication is always about 60–70% of encryption/decryption, and in contrast to that, in case of our scheme, the cost for authentication is about half of the corresponding value for hash tree.

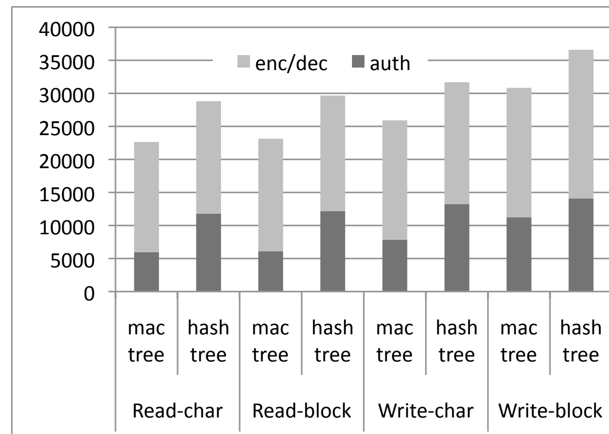

Figure 5: Microbenchmark result, in 10^6 cycles

7. CONCLUSION

In this paper, we proposed a new file encryption scheme for distributed cryptographic file system. Our scheme is based on a new MAC tree construction which uses a universal-hash based MAC scheme. We defined confidentiality and integrity for file encryption schemes, and showed that our scheme satisfies these security goals. We compared the performance of our scheme with Merkle hash tree based schemes, and the experiments show that our scheme provides integrity protection at a smaller cost than Merkle hash tree based schemes.

There are some potential performance enhancements that we didn't pursue in this paper. For example, our scheme uses CTR mode for encryption, and expecting a sequential read, the pseudorandom sequences of CTR could be precomputed [21]. Also, most universal hash functions, including Poly1305 that we use, allow incremental updates. When only a small part of a message changes, this could speed up the MAC computation.

We believe that choices of caching strategy, MAC tree update schedule, arity of the MAC tree, and method for storing the tree could affect the performance of our scheme considerably, but admittedly our choices in these matters were not very systematic. In ongoing work we'll explore these possibilities more thoroughly.

Acknowledgements

This research was supported, in part, by the US National Science Foundation (grant nos. CCF-0621462 and CNS-0448423).

8. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer, *FARSITE: Federated, available, and reliable storage for an incompletely trusted environment.* Proceedings of the 5th OSDI, 2002.

[2] Amazon.com, Inc., *Amazon Simple Storage Service (Amazon S3)*, http://aws.amazon.com/s3/

[3] D. J. Bernstein, *The poly1305-AES message-authentication code*, in FSE, H. Gilbert and H. Handschuh, eds., vol. 3557 of Lecture Notes in Computer Science, Springer, 2005, pp. 32–49.

[4] D. J. Bernstein, *Stronger security bounds for Wegman-Carter-Shoup authenticators*, in Proceedings of EUROCRYPT, vol. 3494 of Lecture Notes in Computer Science, Springer, 2005, pp. 164–180.

[5] D. J. Bernstein and T. Lange (eds.), *eBACS: ECRYPT Benchmarking of Cryptographic Systems.* http://bench.cr.yp.to

[6] Russell Coker, *Bonnie++ File System Benchmark*, http://www.coker.com.au/bonnie++/

[7] G. Brassard, *On computationally secure authentication tags requiring short secret shared keys*, In Advances in Cryptology—CRYPTO 82 (New York, 1983), R. L. Rivest, A. Sherman, and D. Chaum, Eds., Plenum Press, pp. 79–86.

[8] J. L. Carter and M. N. Wegman, *Universal classes of hash functions (extended abstract)*, STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1977, pp. 106–112.

[9] CoreFS, available at http://sourceforge.net/projects/corefs.

[10] K. Fu, *Group sharing and random access in cryptographic storage file systems*, Master's thesis, Massachusetts Institute of Technology, 1999.

[11] K. Fu, F. Kaashoek, D. Mazieres, *Fast and secure distributed read-only file system*, ACM Transactions on Computer Systems, 20:1–24, 2002.

[12] FUSE, available at http://fuse.sourceforge.net/.

[13] K. Gjøsteen, *Security notions for disk encryption*, ESORICS, Lecture Notes in Computer Science, vol. 3679, Springer, 2005, pp. 455–474.

[14] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, *Plutus: Scalable secure file sharing on untrusted storage*, in Proc. Second USENIX Conference on File and Storage Technologies (FAST), 2003.

[15] T. Krovetz, Fast cryptography. Webpage: http://fastcrypto.org/

[16] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, *Secure untrusted data repository (SUNDR)*, in OSDI, 2004, pp. 121–136.

[17] D. Mazieres, M. Kaminsky, M. Kaashoek, E. Witchel, *Separating key management from file system security,* In Proc. 17th ACM Symposium on Operating Systems Principles (SOSP), pp. 124–139, ACM Press, 1999.

[18] D. Mazières and D. Shasha, *Building secure file systems out of byzantine storage*, in PODC, 2002, pp. 108–117.

[19] A. Oprea and M. K. Reiter, *Integrity checking in cryptographic file systems with constant trusted storage*, in Proceedings of the 16th USENIX Security Symposium, 2007, pp. 183–198.

[20] R. Pletka and C. Cachin, *Cryptographic security for a high-performance distributed file system*, in Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies, IEEE Computer Society, 2007, pp. 227–232.

[21] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, *High efficiency counter mode security architecture via prediction and precomputation*, SIGARCH Comput. Archit. News, 33 (2005), pp. 14–24.

[22] V. Shoup, *On fast and provably secure message authentication based on universal hashing*, in Proceedings of Crypto, 1996, pp. 313–328.

[23] M. N. Wegman and L. Carter, *New classes and applications of hash functions*, in FOCS, IEEE, 1979, pp. 175–182.

APPENDIX

A. SKETCH OF SECURITY PROOFS

Here we give informal description of our security theorems and sketches of the proofs. The complete proofs will be on the full version of the paper.

LEMMA 1. *Suppose that the underlying stateful MAC is secure. Then, for any attacker A of Φ, the probability that A ever makes the underlying MAC to repeat a nonce for two tag generation queries is negligible. That is, the chance that A ever makes, via its file operation requests, two tag generation queries $\mathcal{M}_{K_a,K_h}(\langle i,j,N\rangle_a, D)$ and $\mathcal{M}_K(\langle i,j,N\rangle_a, D')$ for some distinct D and D', is negligible.*

PROOF. (Sketch) First note that, if A followed the protocol faithfully, that is, never provided altered $\mathcal{S}$ with its requests, then A cannot ever repeat a nonce: for any i and j, $N_i^{(j)}$ will be properly incremented so each nonce $\langle i,j,N_i^{(j)}\rangle_a$ will be used only once.

Suppose that A succeeds repeating a nonce during an Update or Append request (note that tag generation occurs only for Update or Append requests). Consider the first such successful request. Let's assume that it was an Update request; the case of an Append request can be handled similarly. Because it was the first request during which A repeats a nonce, previously A behaved as a nonce-respecting attacker with respect to the underlying MAC. During the process of the Update request, A first produces l tag verification queries, and then, only if all of the verification queries are passed, A produces $l+1$ tag generation queries. Suppose that during the verification queries of form $\bar{T}_{i_j}^{(j)} \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle i_j, j, \bar{N}_{i_j}^{(j)}\rangle_a, \bar{D}_{i_j}^{(j)})$, no MAC forgery was successful. (Here $\bar{T}_i^{(j)}$, $\bar{N}_i^{(j)}$, $\bar{D}_i^{(j)}$ represents the values from $(t,s) \in \mathcal{T} \times \mathcal{S}$, where $s \in \mathcal{S}$ could be potentially altered by the attacker A. On the other hand, we represent the 'correct' values from the output state of the last successful request as $T_i^{(j)}$, $N_i^{(j)}$, $D_i^{(j)}$.)

In that case, from $j = l$, we see that $(\langle 1, l, \bar{N}_1^{(l)}\rangle_a, \bar{D}_1^{(l)})$ is not forgery. Since $\bar{N}_1^{(j)}$ is from $\mathcal{T}$, we know that $\bar{N}_1^{(j)} = N_1^{(j)}$ is from the previous successful update or append query, and used only once, in order to authenticate $D_1^{(l)}$. Since $(\langle 1, l, \bar{N}_1^{(l)}\rangle_a, \bar{D}_1^{(l)})$ is not forgery, it follows that $\bar{D}_1^{(l)} = D_1^{(l)}$. But $D_1^{(l)}$ contains $N_{i_{l-1}}^{(l-1)}$, and from this it follows that $\bar{N}_{i_{l-1}}^{(l-1)} = N_{i_{l-1}}^{l-1}$. Again, $N_{i_{l-1}}^{l-1}$ was used only once in order to authenticate $D_{i_{l-1}}^{l-1}$, and so on. Therefore we conclude that all of $\bar{N}_{i_j}^{(j)}$ $(j = 0, \ldots, l)$ are so far properly incremented and

used only once. Since the $l+1$ MAC generation queries use nonces of form $\langle i_j, j, \bar{N}_{i_j}^{(j)}+1\rangle_a$, it follows that during this Update request, A failed to repeat any nonce, which is contradiction.

Therefore, during one of the l tag verification queries, A must have made a forgery. But, until then A was a nonce-respecting attacker of $\mathcal{M}$, therefore such probability is negligible. □

Remark 1. *The proof of Lemma 1 applies also for a confidentiality attacker which makes requests of form* Update(i, D_0, D_1)*, and* Append(D_0, D_1)*: the existence of the left-or-right oracle affects only the leaf nodes, but nonce-respecting is related to the update counters, which are above the leaf nodes.*

Theorem 1. *For any efficient attacker A of Φ, its advantage* $\mathrm{Adv}_\Phi^{\mathrm{int}}(A)$ *is negligible, if the underlying MAC is secure.*

Proof. (Sketch) From the previous Lemma 1, we see that except negligible probability, A behaves as a nonce-respecting attacker of the underlying MAC $\mathcal{M}$. Therefore assume that A never repeats a nonce. Then, because $\mathcal{M}$ is secure, the probability that A succeeds making a forgery is negligible. So assume that A never succeeds a forgery.

Suppose that A makes a successful Read(k) request, and receives $\hat{D}$ which is different from D_k, the correct value from the content of Φ at the moment he makes that Read(k) request. Consider the first such read request. Since the read request is successful, all of the verification queries $\bar{T}_{i_j}^{(j)} \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle i_j, j, \bar{N}_{i_j}^{(j)}\rangle_a, \bar{D}_{i_j}^{(j)})$ was successful for $j = 1, \ldots, l$. But from the case $j = l$, $\bar{N}_1^{(l)}$ is read from $\mathcal{T}$, therefore $\bar{N}_1^{(l)} = N_1^{(l)}$, and we know that the nonce $\langle 1, l, N_1^{(l)}\rangle_a$ was used exactly once to authenticate $D_1^{(l)}$. Since A does not make any forgery, it follows that $\bar{D}_1^{(l)} = D_1^{(l)}$. But since $\bar{N}_{i_{l-1}}^{(l-1)}$ is part of $\bar{D}_{i_l}^{(l)} = \bar{D}_1^{(l)}$, we get $\bar{N}_{i_{l-1}}^{(l-1)} = N_{i_{l-1}}^{(l-1)}$. Repeating this, we get $\bar{N}_{i_j}^{(j)} = N_{i_j}^{(j)}$ for $j = 0, \ldots, l$. Finally, since $\bar{T}_{i_0} \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle i_0, 0, \bar{N}_{i_0}^{(0)}\rangle_a, \bar{C}_{i_0})$ is successful but not a forgery, $\bar{C}_{i_0} = C_{i_0} = C_k$, which is the ciphertext block of the correct kth content block D_k. But this contradicts the assumption that $\hat{D} \neq D_k$. Therefore, we conclude that except the negligible event of successful forgery attack of the underlying MAC, A cannot succeed in attacking the integrity of Φ. □

Theorem 2. *For any efficient attacker A of Φ, its advantage* $\mathrm{Adv}_\Phi^{\mathrm{conf}}(A)$ *is negligible, if the underlying MAC is secure.*

Proof. (Sketch) Again, we may assume that A never repeats nonces for the underlying MAC $\mathcal{M}$, and also A never makes successful forgery of $\mathcal{M}$. Consider any Read, Update, or Append request which failed at least one of the tag verification queries

$$T_{i_j}^{(j)} \stackrel{?}{=} \mathcal{M}_{K_a,K_h}(\langle i_j, j, N_{i_j}^{(j)}\rangle_a, D_{i_j}^{(j)})$$

for $j = 1, \ldots, l$. Note that for $j > 0$, the 'message' $D_{i_j}^{(j)}$ consists of only counters $N_i^{(j-1)}$, and does not depend on the kth file block D_k at all, therefore they does not give any information about D_k. If we exclude these file operation requests, from the proof of Theorem 1, we see that the leaf counters $N_k^{(0)}$ can be 'trusted' and they are maintained and updated properly. Therefore, instead of using the leaf counters $N_k^{(0)}$ managed by the MAC tree, we may replace them with trusted and properly incremented counters $N_k^{(0)}$ with only negligible difference in distinguishing probability of the attacker. Therefore, in this modified scenario, the goal of the attacker is to guess the random bit b based on the output of file operation requests, but this time 'correct' and untampered counters $N_k^{(0)}$ are used for the authenticated encryption of the kth file block D_k. Recall that the ciphertext C_k and the authentication tag T_k are as follows:

$$C_k = E_{K_e}(\langle k, 1, N_k^{(0)}\rangle_e) \cdots E_{K_e}(\langle k, d/b, N_k^{(0)}\rangle_e) \oplus D_k, \quad (1)$$

$$T_k = E_{K_a}(\langle k, 0, N_k^{(0)}\rangle_a) \oplus H_{K_h}(C_k), \quad (2)$$

But here D_k and $H_{K_h}(C_k)$ are 'blinded' by pseudorandom sequences generated from fresh nonces, and the attacker is forbidded to make Read(k) request when the 'left' content and the 'right' content of kth block are different. So from C_k, T_k, and $N_k^{(0)}$, the attacker cannot obtain any information about D_k. Therefore, the attacker A cannot do much better than pure guessing in predicting the bit b. □