Integrity Checking in Cryptographic File Systems with Constant Trusted Storage

Alina Oprea[*] **Michael K. Reiter**[†]

November 2006
CMU-CS-06-167

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; alina@cs.cmu.edu
[†]Electrical and Computer Engineering Department and Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA; reiter@cmu.edu

Keywords: cryptographic file systems, integrity checking, Merkle trees.

Abstract

In this paper we propose two new constructions for protecting the integrity of files in cryptographic file systems. Our constructions are designed to exploit two characteristics of many file-system workloads, namely low entropy of file contents and high sequentiality of file block writes. At the same time, our approaches maintain the best features of the most commonly used algorithm today (Merkle trees), including defense against replay attacks of stale (previously overwritten) blocks and a small, constant amount of trusted storage per file. Via implementations in the EncFS cryptographic file system, we evaluate the performance and storage requirements of our new constructions compared to those of Merkle trees. We conclude with guidelines for choosing the best integrity algorithm depending on typical application workload.

1 Introduction

The growth of outsourced storage (e.g., Storage Area Networks) underlines the importance of developing efficient security mechanisms to protect files stored remotely. Cryptographic file systems (e.g., [11, 5, 23, 14, 17, 21, 19]) provide means to protect file secrecy (i.e., prevent leakage of file contents) and integrity (i.e., detect the unauthorized modification of file contents) against the compromise of the file store and attacks on the network while blocks are in transit to/from the file store. Several engineering goals have emerged to guide the design of efficient cryptographic file systems. First, cryptographic protections should be applied at the granularity of individual blocks as opposed to entire files, since the latter requires the entire file to be retrieved to verify its integrity, for example. Second, the application of cryptographic protections to a block should not increase the block size, so as to be transparent to the underlying block store. (Cryptographic protections might increase the number of blocks, however.) Third, the trusted storage required by clients (e.g., for encryption keys and integrity verification information) should be kept to a minimum.

In this paper we propose and evaluate two new algorithms for protecting file integrity in cryptographic file systems. Our algorithms meet these design goals, and in particular implement integrity using only a small constant amount of trusted storage per file. (Of course, as with any integrity-protection scheme, this trusted information for many files could itself be written to a file in the cryptographic file system, thereby reducing the trusted storage costs for many files to that of only one. The need for trusted information cannot be entirely eliminated, however.) In addition, our algorithms exploit two properties of many file-system workloads to achieve efficiencies over prior proposals. First, typical file contents in many file-system workloads have low empirical entropy; such is the case with text files, for example. Our first algorithm builds on a prior proposal that exploits this property [24] and uses tweakable ciphers [20, 15] for encrypting file block contents; this prior proposal, however, did not achieve constant trusted storage per file. Our second algorithm reduces the amount of additional storage needed for integrity by using the fact that a low-entropy block content can be compressed enough to store its hash inside the block. The second property that we exploit in our algorithms is that blocks of the same file are often written sequentially, a characteristic that, to our knowledge, has not been previously utilized.

By designing integrity mechanisms that exploit these properties, we demonstrate more efficient integrity protections in cryptographic file systems than have previously been possible for many workloads. The measures of efficiency that we consider include the amount of untrusted storage required by the integrity mechanism (over and above that required for file blocks); the *integrity bandwidth*, i.e., the amount of this information that must be accessed (updated or read) when accessing a single file block, averaged over all blocks in a file, all blocks in all files, or all accesses in a trace (depending on context); and the average write and read latencies.

The standard against which we compare our algorithms is the Merkle tree [22], which to date is the overwhelmingly most popular method of integrity protection for a file. Merkle trees can be implemented in cryptographic file systems so as to meet the requirements outlined above, in particular requiring trusted storage per file of only one output of a cryptographic hash function (e.g., 20 bytes for SHA-1 [26]). They additionally offer an integrity bandwidth per file that is logarithmic in the number of file blocks. However, Merkle trees are oblivious to file block contents and access characteristics, and we show that by exploiting these, we can generate far more efficient integrity mechanisms for some workloads.

We have implemented our integrity constructions and Merkle trees in EncFS [10], an open-source user-level file system that transparently provides file block encryption on top of FUSE [13]. We provide an evaluation of the three approaches with respect to our measures of interest, demonstrating how file contents, as well as file access patterns, have a great influence on the performance of the three schemes. Our experiments demonstrate that there is not a clear winner among the three constructions for *all* workloads, in that different integrity constructions are best suited to particular workloads. We thus conclude that a cryptographic file system should implement all three schemes and give higher-level applications an option to choose the appropriate integrity mechanism.

2 Random Access Integrity Model

We consider the model of a cryptographic file system that provides random access to files (similar to the NFS file system). Encrypted data is stored on untrusted storage servers and there is a mechanism for distributing the cryptographic keys to authorized parties. A small (on the order of several hundred bytes), fixed-size per file, *trusted storage* is available for authentication data.

We assume that the storage servers are actively controlled by an adversary. The adversary can adaptively alter the data stored on the storage servers or perform any other attack on the stored data, but it cannot modify or observe the trusted storage. A particularly interesting attack that the adversary can mount is a *replay attack*, in which stale data is returned to read requests of clients. Using the trusted storage to keep some constant-size information per file, and keeping more information per file on untrusted storage, our goal is to design and evaluate different integrity algorithms that allow the update and verification of individual blocks in files and that detect data modification and replay attacks.

In our framework, a file F is divided into n fixed-size blocks $B_1 B_2 \dots B_n$ (the last block B_n might be shorter than the first $n-1$ blocks), each encrypted individually and stored on the untrusted storage servers (n differs per file). The constant-size, trusted storage for file F is denoted TS_F; additional storage for file F, which can reside in untrusted storage, is denoted US_F.

The storage interface provides two basic operations to the clients: $F.\mathsf{WriteBlock}(i, c)$ stores content c at block index i in file F and $c \leftarrow F.\mathsf{ReadBlock}(i)$ reads (encrypted) content from block index i in file F. An integrity algorithm for an encrypted file system consists of five operations. In the initialization algorithm $\mathsf{Init}(1^\kappa)$ for file F, the encryption key for the file is generated given as input a security parameter. In an update operation $\mathsf{Update}(i, b)$ for file F, an authorized client updates the i-th block in the file with the encryption of block content b and updates the integrity information for the i-th block stored in TS_F and US_F. In a check operation $\mathsf{Check}(i)$ for file F, an authorized client checks that the (decrypted) block content read from the i-th block in file F is authentic, using the additional storage TS_F and US_F for file F. The check operation return true if the client believes that the block content is authentic and false otherwise. A client can additionally perform an append operation $\mathsf{Append}(b)$ for file F, in which a new block that contains the encryption of b is appended to the encrypted file, and a Delete operation that deletes the last block in a file and updates the integrity information for the file.

In designing an integrity algorithm for an encrypted file system, we consider the following metrics. First is the latency of the Update (or Append) and Check algorithms, which are executed every time a block is written or read from a file. Second is the size of the untrusted storage US_F; we will always enforce that the trusted storage TS_F is of constant size, independent of the number of blocks. Third is the integrity bandwidth for updating and checking individual file blocks, defined as the number of bytes from US_F accessed (updated or read) when accessing a block of file F, averaged over either: all blocks in F when we speak of a per-file integrity bandwidth; all blocks in all files when we speak of the integrity bandwidth of the file system; or all blocks accessed in a particular trace when we speak of one trace.

3 Preliminaries

In this section, we review some preliminary material needed for the integrity constructions described in Section 5.

3.1 Merkle Trees

Merkle trees [22] are used to authenticate n data items with constant-size trusted storage. A Merkle tree for data items $M_1, \ldots, M_n$, denoted $\mathsf{MT}(M_1, \ldots, M_n)$, is a binary tree that has $M_1, \ldots, M_n$ as leaves. An interior node of the tree with children C_L and C_R is the hash of the concatenation of its children (i.e., $h(C_L||C_R)$, for h a collision-resistant hash function). The trusted storage needed to authenticate the n data items is the root of the tree.

We define the Merkle tree for a file F with n blocks $B_1, \ldots, B_n$ to be $\mathsf{MT}_F = \mathsf{MT}(h(1||B_1), \ldots, h(n||B_n))$. A Merkle tree with a given set of leaves can be constructed in multiple ways. In our implementation, we choose to append a new block in the tree as a right-most child, so that the tree has the property that all the left subtrees are complete. We give an example in Figure 1 of a Merkle tree for a file that initially has six blocks: $B_1, \ldots, B_6$. We also show how the Merkle tree for the file is modified when block B_7 is appended to the file.

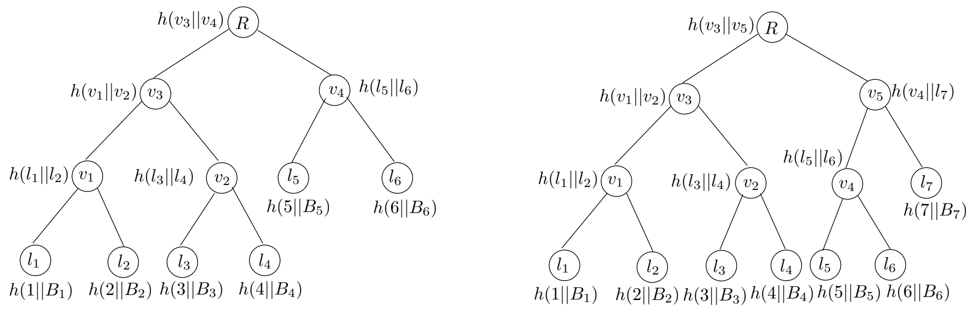

Figure 1: Merkle tree for a file with 6 blocks on the left; after block 7 is appended on the right.

Before describing the algorithms on the Merkle trees, we need to introduce some notation. For a tree T, T.root is the root of the tree, T.no_leaves is the number of leaves in T and T.leaf$[i]$ is the i-th leaf in the tree counting from left to right. For a node v in the tree, v.hash is the hash value stored at the node and for a non-leaf node v, v.left and v.right are pointers to the left and right children of v, respectively. v.sib and v.parent denote the sibling and parent of node v, respectively.

The algorithms UpdatePathRoot, CheckPathRoot, UpdateTree, CheckTree, AppendTree and DeleteTree on the Merkle tree T are described in Figure 2.

```
$T$.UpdatePathRoot(R, $v$):
  while $v \neq T$.root
    $s \leftarrow v$.sib
    if $v$.parent.left $= v$
      $v$.parent.hash $\leftarrow h(v$.hash$||s$.hash$)$
    else
      $v$.parent.hash $\leftarrow h(s$.hash$||v$.hash$)$
    $v \leftarrow v$.parent
  R $\leftarrow T$.root
```

```
$T$.CheckPathRoot(R, $v$):
  $hval \leftarrow v$.hash
  while $v \neq T$.root
    $s \leftarrow v$.sib
    if $v$.parent.left $= v$
      $hval \leftarrow h(hval||s$.hash$)$
    else
      $hval \leftarrow h(s$.hash$||hval)$
    $v \leftarrow v$.parent
  if R $= hval$
    return true
  else
    return false
```

```
$T$.UpdateTree(R, $i$, $hval$):
  if $T$.CheckPathRoot(R, $T$.leaf$[i]$) $=$ true
    $T$.leaf$[i]$.hash $\leftarrow hval$
    $T$.UpdatePathRoot(R, $T$.leaf$[i]$)
```

```
$T$.CheckTree(R, $i$, $hval$):
  if $T$.leaf$[i]$.hash $\neq hval$
    return false
  return $T$.CheckPathRoot(R, $T$.leaf$[i]$)
```

```
$T$.AppendTree(R, $hval$):
  $u$.hash $\leftarrow hval$
  $u$.left $\leftarrow$ null; $u$.right $\leftarrow$ null
  $depth \leftarrow \lceil \log_2(T$.no_leaves$) \rceil$
  $d \leftarrow 0$; $v \leftarrow T$.root
  while $v$.right and $d < depth$
    $v \leftarrow v$.right; $d \leftarrow d + 1$
  if $d = depth$
    $p \leftarrow T$.root
  else
    $p \leftarrow v$.parent
  if $T$.CheckPathRoot(R, $p$) $=$ true
    $p$.left $\leftarrow p$
    $p$.right $\leftarrow u$
    $p$.hash $\leftarrow h(p$.hash$||u$.hash$)$
    $T$.UpdatePathRoot(R, $p$)
```

```
$T$.DeleteTree(R):
  $v \leftarrow T$.root
  while $v$.right
    $v \leftarrow v$.right
  if $v = T$.root
    $T \leftarrow$ null
  else
    $p \leftarrow v$.parent
    if $T$.CheckPathRoot(R, $p$) $=$ true
      if $p = T$.root
        $T$.root $\leftarrow p$.left
      else
        $p$.parent.right $\leftarrow p$.left
      $T$.UpdatePathRoot(R, $p$)
```

Figure 2: The UpdateTree, CheckTree, AppendTree and DeleteTree algorithms for a Merkle tree T.

- In the UpdatePathRoot(R, v) algorithm for tree T, the hashes stored in the nodes on the path from node v to the root of the tree are updated. For that, all the hashes stored in the siblings of those nodes are read. Finally, the hash from the root of T is stored in R.

- In the CheckPathRoot(R, v) algorithm for tree T, the hashes stored in the nodes on the path from node v to the root of the tree are computed, by reading all the hashes stored in the siblings of those nodes. Finally, the hash of the root of T is checked to match the parameter R.

- In the $\mathsf{UpdateTree}(\mathsf{R}, i, hval)$ algorithm for tree T, the hash stored at the i-th leaf of T is updated to $hval$. This triggers an update of all the hashes stored on the path from the i-th leaf to the root of the tree and an update of the value stored in R.

- The $\mathsf{CheckTree}(\mathsf{R}, i, hval)$ algorithm for tree T checks that the hash stored at the i-th leaf matches $hval$. All the hashes stored at the nodes on the path from the i-th leaf to the root are computed and the root of T is checked finally to match the value stored in R.

- Algorithm $\mathsf{AppendTree}(\mathsf{R}, hval)$ for tree T appends a new leaf u that stores the hash value $hval$ to the tree. The right-most node v in the tree is searched first. If the tree is already a complete tree, then the new leaf is added as the right child of the root of the tree and the existing tree becomes the left child of the new root. Otherwise, the new leaf is appended as the right child of the parent p of v, and the subtree rooted at p becomes the left child of p. This way, for any node in the Merkle tree, the left subtree is complete and the tree determined by a set of leaves is unique.

- The $\mathsf{DeleteTree}(\mathsf{R})$ algorithm for tree T deletes the last leaf from the tree. First, the right-most node v in the tree is searched. If this node is the root itself, then the tree becomes null. Otherwise, node v is removed from the tree and the subtree rooted at the sibling of v is moved in the position of v's parent.

3.2 Encryption Schemes and Tweakable Ciphers

An encryption scheme consists of a key generation algorithm Gen that takes as input a security parameter and outputs an encryption key, an encryption algorithm $\mathsf{E}_k(m)$ that outputs the encryption of a message m with secret key k and a decryption algorithm $\mathsf{D}_k(c)$ that outputs the decryption of a ciphertext c with secret key k. A widely used secure encryption scheme is AES [2] in CBC mode [8].

A tweakable cipher [20, 15] is, informally, a length-preserving encryption method that uses a *tweak* in both the encryption and decryption algorithms for variability. A tweakable encryption of a message m with tweak t and secret key k is denoted $\mathsf{E}_k^t(m)$ and, similarly, the decryption of ciphertext c with tweak t and secret key k is denoted $\mathsf{D}_k^t(m)$. The tweak is a public parameter, and the security of the tweakable cipher is based only on the secrecy of the encryption key. Tweakable ciphers can be used to encrypt fixed-size blocks written to disks in a file system. Suitable values of the tweak for this case are, for example, block addresses or block indices in the file.

The security of tweakable ciphers implies an interesting property, called *non-malleability* [15], that guarantees that if only a single bit is changed in a valid ciphertext, then its decryption is indistinguishable from a random plaintext. Tweakable cipher constructions include CMC [15] and EME [16].

3.3 Efficient Block Integrity Using Randomness of Block Contents

Oprea et al. [24] provide an efficient integrity construction in a block-level storage system. This integrity construction is based on the experimental observation that contents of blocks written to disk usually are efficiently distinguishable from random blocks (i.e., we denote by *random block* a block uniformly chosen at random from the set of all blocks of a fixed length). Assuming that data blocks are encrypted with a tweakable cipher, the integrity of the blocks that are efficiently distinguishable from random blocks can be checked by performing a randomness test on the block contents. The non-malleability property of tweakable ciphers implies that if block contents after decryption do not look random (i.e., have low entropy), then it is very likely that the contents are authentic. This idea permits a reduction in the trusted storage needed for checking block integrity: a hash is stored only for those (usually few) blocks that are indistinguishable from random blocks (or, in short, *random-looking blocks*).

An example of a statistical test `IsRand` [24], which can be used to distinguish block contents from random-looking blocks, evaluates the entropy of a block and considers random those blocks that have an entropy higher than a threshold chosen experimentally. For a block B, $\texttt{IsRand}(B)$ returns 1 with high probability if B is a uniformly random block in the block space and 0, otherwise.

We use the ideas from Oprea et al. [24] as a starting point for our first algorithm for implementing file integrity with constant trusted storage per file. Our second algorithm here, however, exploits low-entropy of files in a different way. Both of our constructions could be further applied to other authentication problems that have the restriction of using constant-size trusted storage, such as memory integrity checking (e.g., [7]).

4 Write Counters for File Blocks

All the integrity constructions for encrypted storage described in the next section use write counters for the blocks in a file. A write counter for a block denotes the total number of writes done to that block index. Counters are used to reduce the additional storage space taken by encrypting with a block cipher in CBC mode. Counters are also a means of distinguishing different writes performed to the same block address and as such, can be used to prevent against replay attacks.

We define several operations for the write counters of the blocks in a file F:

- The $\mathsf{update\_ctr}(i)$ algorithm either initializes the value of the counter for the i-th block in file F with 1, or it increments the counter for the i-th block if it has already been initialized.
- Function $\mathsf{get\_ctr}(i)$ returns the value of the counter for the i-th block in file F.
- Algorithm $\mathsf{del\_ctr}(i)$ deletes the counter for the i-th block in file F.

Operations $\mathsf{update\_ctr}$ and $\mathsf{del\_ctr}$ also update the information for the counters stored in US_F. When counters are used for encryption, they can be safely stored in the untrusted storage space for a file. However, in the case in which counters protect against replay attacks, they need to be authenticated with a small amount of trusted storage. We define two algorithms for authenticating block write counters, both of which are invoked by an authorized client:

- Algorithm $\mathsf{auth\_ctr}$ modifies the trusted storage space TS_F of file F to contain the trusted authentication information for the write counters of F.
- Algorithm $\mathsf{check\_ctr}$ checks the authenticity of the counters stored in US_F using the trusted storage TS_F for file F and returns true if the counters are authentic and false, otherwise.

4.1 Length-Preserving Stateful Encryption with Counters

Secure encryption schemes are usually not length-preserving. However, one of our design goals stated in the introduction is to add security (and, in particular, encryption) to file systems in a manner transparent to the storage servers. For this purpose, we introduce here the notion of *length-preserving stateful encryption scheme* for a file F, an encryption scheme that constructs encrypted blocks that preserve the length of original blocks, and stores any additional information in the untrusted storage space for the file. We define a length-preserving stateful encryption scheme for a file F to consist of a key generation algorithm $\mathsf{G}^{\mathsf{len}}$ that generates an encryption key for the file given a security parameter, an encryption algorithm $\mathsf{E}^{\mathsf{len}}$ that encrypts block content b for block number i with key k and outputs ciphertext c, and a decryption algorithm $\mathsf{D}^{\mathsf{len}}$ that decrypts the encrypted content c of block i with key k and outputs the plaintext b. Both the $\mathsf{E}^{\mathsf{len}}$ and $\mathsf{D}^{\mathsf{len}}$ algorithms also modify the untrusted storage space for the file.

Write counters for file blocks can be used to construct a length-preserving stateful encryption scheme. Let $(\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ be an encryption scheme constructed from a block cipher in CBC mode. To encrypt a n-block message in the CBC encryption mode, a random initialization vector is chosen. The ciphertext consists of $n + 1$ blocks, with the first being the initialization vector. We denote by $\mathsf{E}_k(b, iv)$ the output of the encryption of b (excluding the first block) using key k and initialization vector iv, and similarly by $\mathsf{D}_k(c, iv)$ the decryption of c using key k and initialization vector iv.

We replace the random initialization vectors for encrypting a block in the file in CBC mode with a pseudorandom function application of the block index concatenated with the write counter for the block. This is intuitively secure because different initialization vectors are used for different encryptions of the same block, and moreover, the properties of pseudorandom functions imply that the initialization vectors are indistinguishable from random. It is thus enough to store the write counters for the blocks of a file, and the initialization vectors for the file blocks can be easily inferred.

The $\mathsf{G}^{\mathsf{len}}$, $\mathsf{E}^{\mathsf{len}}$ and $\mathsf{D}^{\mathsf{len}}$ algorithms for a file F are described in Figure 3. Here κ is the security parameter and $G : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{K}$ denotes a pseudorandom function family with key space $\mathcal{K}$ and message space $\mathcal{C}$, which is the set of all block indices concatenated with block counter values.

4.2 Storage and Authentication of Block Write Counters

A problem that needs to be addressed in the design of the various integrity algorithms described below is the storage and authentication of the block write counters. If a counter per file block were used, this would result in significant additional storage for counters. We investigate more efficient

```
$F.\mathsf{G}^{\mathsf{len}}(1^\kappa)$:
  $k_1 \xleftarrow{R} \mathcal{K}$
  $k_2 \leftarrow \mathsf{Gen}(1^\kappa)$
  return $k = k_1 || k_2$
```

```
$F.\mathsf{E}_k^{\mathsf{len}}(i, b)$:
  parse $k$ as $k_1 || k_2$
  $F.\mathsf{update\_ctr}(i)$
  $iv \leftarrow G_{k_1}(i || F.\mathsf{get\_ctr}(i))$
  $c \leftarrow \mathsf{E}_{k_2}(b, iv)$
  return $c$
```

```
$F.\mathsf{D}_k^{\mathsf{len}}(i, c)$:
  parse $k$ as $k_1 || k_2$
  $iv \leftarrow G_{k_1}(i || F.\mathsf{get\_ctr}(i))$
  $b \leftarrow \mathsf{D}_{k_2}(c, iv)$
  return $b$
```

Figure 3: Implementing a length-preserving stateful encryption scheme with write counters.

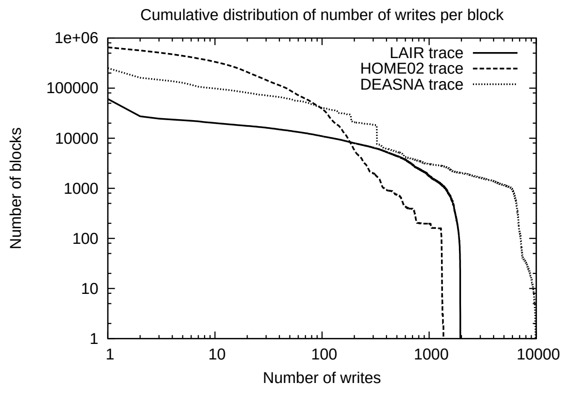

Figure 4: Cumulative distribution of number of writes per block

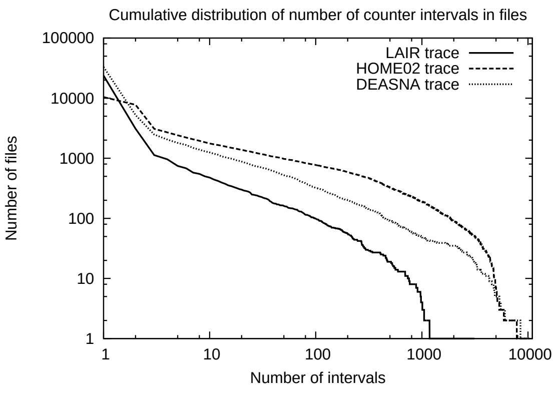

Figure 5: Cumulative distribution of number of counter intervals in files

methods of storing the block write counters, based on analyzing the file access patterns in NFS traces collected at Harvard University [9].

Counter intervals. We performed some experiments on the NFS Harvard traces [9] in order to analyze the file access patterns. We considered three different traces (LAIR, DEASNA and HOME02) for a period of one week. The LAIR trace consists from research workload traces from Harvard's computer science department. The DEASNA trace is a mix of research and email workloads from the division of engineering and applied sciences at Harvard. HOME02 is mostly the email workload from the campus general purpose servers.

We first plotted the cumulative distribution of the number of writes per block for each trace in Figure 4 in a log-log scale (i.e., for x number of writes on the x-axis, the y axis represents the number of blocks that have been written at least x times).

Ellard et al. [9] make the observation that a large number of file accesses are sequential. This leads to the idea that the values of the write counters for adjacent blocks in a file might be correlated. To test this hypothesis, we represent counters for blocks in a file using *counter intervals*. A counter interval is defined as a sequence of consecutive blocks in a file that all share the same value of the write counter. For a counter interval, we need to store only the beginning and end of the interval, and the value of the write counter. We plot the cumulative distribution of the number of counter intervals per files for the three Harvard traces in Figure 5 (i.e., for x number of intervals on the x-axis, the y axis represents the number of files that have at least x counter intervals).

Figure 5 validates the hypothesis that for the large majority of files in the three traces considered, a small number of counter intervals needs to be stored and only for few files the number of counter intervals is large (i.e., over 1000). The average storage per file using counter intervals is several orders of magnitude smaller than that used by storing a counter per block, as shown in Table 1. This justifies our design choice to use counter intervals for representing counter values in the integrity algorithms presented in the next section.

	LAIR	DEASNA	HOME02
Counter per block	1.79 MB	2.7 MB	8.97 MB
Counter intervals	5 bytes	9 bytes	163 bytes

Table 1: Average storage per file for two counter representation methods.

Counter representation. The counter intervals for file F are represented by two arrays: $\mathsf{IntStart}_F$ keeps the block indices where new counter intervals start and CtrVal_F keeps the values of the write counter for each interval. The trusted storage TS_F for file F includes either the arrays $\mathsf{IntStart}_F$ and CtrVal_F if they fit into TS_F or their hashes, otherwise. In the limit, to reduce the bandwidth for integrity, we could build a Merkle tree to authenticate each of these arrays and store the root of these trees in TS_F, but we have not seen in the Harvard traces files that would warrant this.

We omit here the implementation details for the update_ctr, get_ctr and del_ctr operations on counters (which are immediate), but describe the algorithms for authenticating counters with a constant amount of trusted storage. Assume that the length of available trusted storage for counters for file F is $\mathsf{L}_{\mathsf{ctr}}$. For an array A, A.size is the number of bytes needed for all the elements in the array and $h(\mathsf{A})$ is the hash of concatenated elements in the array. We also store in trusted storage a flag ctr-untr whose value is true if the counter arrays $\mathsf{IntStart}_F$ and CtrVal_F are stored in the untrusted storage space of F and false otherwise. The auth_ctr and check_ctr algorithms are described in Figure 6.

F.auth_ctr():	F.check_ctr():
if $\mathsf{IntStart}_F$.size + CtrVal_F.size > $\mathsf{L}_{\mathsf{ctr}}$	if ctr-untr= true
store $h_1 = h(\mathsf{IntStart}_F)$ and $h_2 = h(\mathsf{CtrVal}_F)$ in TS_F	get $\mathsf{IntStart}_F$ and CtrVal_F from US_F
store $\mathsf{IntStart}_F$ and CtrVal_F in US_F	get h_1 and h_2 from TS_F
ctr-untr = true	if $h_1 = h(\mathsf{IntStart}_F)$ and $h_2 = h(\mathsf{CtrVal}_F)$
else	return true
store $\mathsf{IntStart}_F$ and CtrVal_F in TS_F	else
ctr-untr = false	return false

Figure 6: The auth_ctr and check_ctr algorithms for counter intervals.

If the counter intervals for a file get too dispersed, then the size of the arrays $\mathsf{IntStart}_F$ and CtrVal_F might increase significantly. To keep the untrusted storage for integrity low, we could periodically change the encryption key for the file, re-encrypt all blocks in the file, and reset the block write counters to 0.

5 Integrity Constructions for Encrypted Storage

In this section, we first present a Merkle tree integrity construction for encrypted storage, used in file systems such as Cepheus [11], FARSITE [1], and Plutus [17]. Second, we introduce a new integrity construction based on tweakable ciphers that uses some ideas from Oprea et al. [24]. Third, we give a new construction based on compression levels of block contents. We evaluate the performance of the integrity algorithms described here in Section 7.

5.1 The Merkle Tree Construction $\mathsf{MT\text{-}EINT}$

In this construction, file blocks can be encrypted with any length-preserving stateful encryption scheme and they are authenticated with a Merkle tree. More precisely, if $F = B_1 \dots B_n$ is a file with n blocks, then the untrusted storage for integrity for file F is $\mathsf{US}_F = \mathsf{MT}(h(1||B_1), \dots, h(n||B_n))$ (for h a collision-resistant hash function), and the trusted storage TS_F is the root of this tree.

The algorithm Init runs the key generation algorithm $F.\mathsf{G}^{\mathsf{len}}(1^\kappa)$ of the length-preserving stateful encryption scheme for file F. The algorithms Update, Check, Append and Delete of the $\mathsf{MT\text{-}EINT}$ construction are given in Figure 7. We denote here by $F.\mathsf{enc\_key}$ the encryption key for file F (generated in the Init algorithm) and $F.\mathsf{blocks}$ the number of blocks in file F. In the $\mathsf{Update}(i, b)$ algorithm for file F, the i-th leaf in MT_F is updated with the hash of the new block content using the algorithm $\mathsf{UpdateTree}$ and the encryption of b is stored in the i-th block of F. In the $\mathsf{Check}(i)$ algorithm for file F, the i-th block is read and decrypted, and its integrity is checked using the $\mathsf{CheckTree}$ algorithm. To append a new block b to file F with algorithm $\mathsf{Append}(b)$, a new leaf is appended to MT_F with the algorithm $\mathsf{AppendTree}$, and then an encryption of b is stored in the $(n+1)$-th block of F (for n the number of blocks of F). To delete the last block from a file F with algorithm Delete, the last leaf in MT_F is deleted with the algorithm $\mathsf{DeleteTree}$.

```
$F.\mathsf{Update}(i, b)$:
  $k \leftarrow F.\mathsf{enc\_key}$
  $\mathsf{MT}_F.\mathsf{UpdateTree}(\mathsf{TS}_F, i, h(i||b))$
  $c \leftarrow F.\mathsf{E}^{\mathsf{len}}_k(i, b)$
  $F.\mathsf{WriteBlock}(i, c)$
```

```
$F.\mathsf{Check}(i)$:
  $k \leftarrow F.\mathsf{enc\_key}$
  $\bar{B}_i \leftarrow F.\mathsf{ReadBlock}(i)$
  $B_i \leftarrow F.\mathsf{D}^{\mathsf{len}}_k(i, \bar{B}_i)$
  return $\mathsf{MT}_F.\mathsf{CheckTree}(\mathsf{TS}_F, i, h(i||B_i))$
```

```
$F.\mathsf{Append}(b)$:
  $k \leftarrow F.\mathsf{enc\_key}$
  $n \leftarrow F.\mathsf{blocks}$
  $\mathsf{MT}_F.\mathsf{AppendTree}(\mathsf{TS}_F, h(n+1||b))$
  $c \leftarrow F.\mathsf{E}^{\mathsf{len}}_k(n+1, b)$
  $F.\mathsf{WriteBlock}(n+1, c)$
```

```
$F.\mathsf{Delete}()$:
  $n \leftarrow F.\mathsf{blocks}$
  $\mathsf{MT}_F.\mathsf{DeleteTree}(\mathsf{TS}_F)$
  delete $B_n$ from file $F$
```

Figure 7: The Update, Check, Append and Delete algorithms for the $\mathsf{MT\text{-}EINT}$ construction.

The $\mathsf{MT\text{-}EINT}$ construction detects data modification attacks, as file block contents are authenticated by the root of the Merkle tree for each file. Block swapping attacks are prevented by including the block indices in the hashes stored in the leaves of the Merkle trees. The $\mathsf{MT\text{-}EINT}$ construction is also secure against replay attacks, as the tree contains the hashes of the latest version of the data blocks and the root of the Merkle tree is authenticated in trusted storage.

5.2 The Randomness Test Construction RAND-EINT

Whereas in the Merkle tree construction any length-preserving stateful encryption algorithm can be used to individually encrypt blocks in a file, the randomness test construction uses the observation from [24] that the integrity of the blocks that are efficiently distinguishable from random blocks can be checked with a randomness test if a tweakable cipher is used to encrypt them. As such, integrity information is stored only for random-looking blocks.

In this construction, a Merkle tree per file that authenticates the contents of the random-looking blocks is built. The untrusted storage for integrity US_F for file $F = B_1 \dots B_n$ includes this tree $\mathsf{RTree}_F = \mathsf{MT}(h(i||B_i) : i \in \{1, \dots, n\}$ and $\mathtt{IsRand}(B_i) = 1)$, and, in addition, the set of block numbers that are random-looking $\mathsf{RArr}_F = \{i \in \{1, \dots, n\} : \mathtt{IsRand}(B_i) = 1\}$, ordered the same as the leaves in the previous tree RTree_F. The root of the tree RTree_F is kept in the trusted storage TS_F for file F.

To prevent against replay attacks, clients need to distinguish different writes of the same block in a file. A simple idea [24] is to use a counter per file block that denotes the number of writes of that block, and make the counter part of the encryption tweak. The block write counters need to be authenticated in the trusted storage space for the file F to prevent clients from accepting valid older versions of a block that are considered not random by the randomness test. To ensure that file blocks are encrypted with different tweaks, we define the tweak for a block to be the hash of the block number concatenated with the block write counter. The properties of tweakable ciphers imply that if a block is decrypted with a different counter, then it will look random with high probability.

The algorithm Init selects a key at random from the key space of the tweakable encryption scheme E. The Update, Check, Append and Delete algorithms of RAND-EINT are detailed in Figure 8. For the array RArr_F, RArr_F.items denotes the number of items in the array, RArr_F.last denotes the last element in the array, and the function $\mathsf{SearchOffset}(i)$ for the array RArr_F gives the position in the array where index i is stored (if it exists in the array).

- In the $\mathsf{Update}(i, b)$ algorithm for file F, the write counter for block i is incremented and the counter authentication information from TS_F is updated with the algorithm auth_ctr. Then, the randomness test `IsRand` is applied to block content b. If b is not random looking, then the leaf corresponding to block i (if it exists) has to be removed from RTree_F. This is done with the algorithm DelOffsetTree, described in Figure 9. On the other hand, if b is random-looking, then the leaf corresponding to block i has to be either updated with the new hash (if it exists in the tree) or appended in RTree_F. Finally, the tweakable encryption of b is stored in the i-th block of F.

- In the $\mathsf{Check}(i)$ algorithm for file F, the authentication information from TS_F for the block counters is checked first. Then the i-th block of F is read and decrypted, and checked for integrity. If the content of the i-th block is not random-looking, then by the properties of tweakable ciphers we can infer that the block is valid. Otherwise, the integrity of the i-th block is checked using the tree RTree_F. If i is not a block number in the tree, then the integrity of block i is unconfirmed and the block is rejected.

```
$F$.Update$(i, b)$ :
  $k \leftarrow F$.enc_key
  $F$.update_ctr$(i)$
  $F$.auth_ctr$()$
  if $\texttt{IsRand}(b) = 0$
    if $i \in \mathsf{RArr}_F$
      $\mathsf{RTree}_F$.DelOffsetTree$(\mathsf{TS}_F, \mathsf{RArr}_F, i)$
  else
    if $i \in \mathsf{RArr}_F$
      $j \leftarrow \mathsf{RArr}_F$.SearchOffset$(i)$
      $\mathsf{RTree}_F$.UpdateTree$(\mathsf{TS}_F, j, h(i||b))$
    else
      $\mathsf{RTree}_F$.AppendTree$(\mathsf{TS}_F, h(i||b))$
      append $i$ at end of $\mathsf{RArr}_F$
  $F$.WriteBlock$(i, \mathsf{E}_k^{h(i||F.\mathsf{get\_ctr}(i))}(b))$
```

```
$F$.Check$(i)$:
  $k \leftarrow F$.enc_key
  if $F$.check_ctr$()$ = false
    return false
  $\bar{B}_i \leftarrow F$.ReadBlock$(i)$
  $B_i \leftarrow \mathsf{D}_k^{h(i||F.\mathsf{get\_ctr}(i))}(\bar{B}_i)$
  if $\texttt{IsRand}(B_i) = 0$
    return true
  else
    if $i \in \mathsf{RArr}_F$
      $j \leftarrow \mathsf{RArr}_F$.SearchOffset$(i)$
      return $\mathsf{RTree}_F$.CheckTree$(\mathsf{TS}_F, j, h(i||B_i))$
    else
      return false
```

```
$F$.Append$(b)$:
  $k \leftarrow F$.enc_key
  $F$.update_ctr$(n+1)$
  $F$.auth_ctr$()$
  $n \leftarrow F$.blocks
  if $\texttt{IsRand}(b) = 1$
    $\mathsf{RTree}_F$.AppendTree$(\mathsf{TS}_F, h(n+1||b))$
    append $n+1$ at end of $\mathsf{RArr}_F$
  $F$.WriteBlock$(n+1, \mathsf{E}_k^{h(n+1||F.\mathsf{get\_ctr}(n+1))}(b))$
```

```
$F$.Delete$()$:
  $F$.del_ctr$(n)$
  $F$.auth_ctr$()$
  $n \leftarrow F$.blocks
  if $\texttt{IsRand}(B_n) = 1$
    $\mathsf{RTree}_F$.DelOffsetTree$(\mathsf{TS}_F, \mathsf{RArr}_F, n)$
  delete $B_n$ from file $F$
```

Figure 8: The Update, Check, Append and Delete algorithms for the RAND-EINT construction.

```
$T$.DelOffsetTree$(\mathsf{TS}_F, \mathsf{RArr}_F, i)$:
  $j \leftarrow \mathsf{RArr}_F$.SearchOffset$(i)$
  $l \leftarrow \mathsf{RArr}_F$.last
  if $j = l$
    $T$.DeleteTree$(\mathsf{TS}_F)$
  else
    $T$.UpdateTree$(\mathsf{TS}_F, j, h(l||B_l))$
    $\mathsf{RArr}_F[j] \leftarrow l$
    $\mathsf{RArr}_F$.items $\leftarrow \mathsf{RArr}_F$.items $- 1$
```

Figure 9: The DelOffsetTree algorithm for a tree T deletes the hash of block i from T and moves the last leaf in its position, if necessary, to not allow holes in the tree.

- When a new block b is appended to file F with algorithm Append(b), the write counter for the new block is initialized and the authentication information for counters updated. Furthermore, the hash of the block content is added to RTree_F only if the block is random-looking. In addition, the index $n+1$ (where n is the current number of blocks in F) is added to RArr_F in this case. Finally, the tweakable encryption of b is stored in the $(n+1)$-th block of F.

- To delete the last block in file F with algorithm Delete, the write counter for the last block is deleted and the authentication information for the counters is updated. Then, the last block of F is read and decrypted. If the block is random-looking, then its hash has to be removed from tree RTree_F, using the algorithm DelOffsetTree.

It is not necessary to authenticate in trusted storage the array RArr_F of indices of the random-looking blocks in a file. The reason is that the root of RTree_F is authenticated in trusted storage and this implies that an adversary cannot modify the order of the leaves in RTree_F without being detected in the AppendTree, UpdateTree or CheckTree algorithms.

The construction RAND-EINT protects against unauthorized modification of data written to disk and block swapping attacks by authenticating the root of RTree_F in the trusted storage space for each file. By using write counters in the encryption of block contents and authenticating the values of the counters in trusted storage, this construction provides defense against replay attacks and provides all the security properties of the MT-EINT construction.

5.3 The Compress-and-Hash Construction COMP-EINT

This construction is again based on the intuition that many workloads feature low-entropy files, but attempts to exploit this in a different way. In this construction, the block is compressed before encryption. If the compression level of the block content is high enough, then the hash of the block can be stored in the block itself, reducing the amount of storage necessary for integrity. Like in the previous construction, a Merkle tree RTree_F is built over the hashes of the blocks in file F that cannot be compressed enough, and the root of the tree is kept in trusted storage. In order to prevent replay attacks, it is necessary that block write counters are hashed together with block contents. Similarly to scheme RAND-EINT, the write counters for a file F need to be authenticated in the trusted storage space TS_F (see Section 4.2 for a description of the algorithms auth_ctr and check_ctr used to authenticate and check the counters, respectively).

In the description of the integrity scheme, we assume we are given a compression algorithm compress and a decompression algorithm decompress such that $\mathsf{decompress}(\mathsf{compress}(m)) = m$, for any message m. We can also pad messages up to a certain fixed length by using the pad function with an output of l bytes, and unpad a padded message with the unpad function such that $\mathsf{unpad}(\mathsf{pad}(m)) = m$, for all messages m of length less than l bytes. We can use standard padding methods for implementing these algorithms [3].

The algorithm Init runs the key generation algorithm $F.\mathsf{G}^{\mathsf{len}}(1^\kappa)$ of the length-preserving stateful encryption scheme for file F. The Update, Append, Check and Delete algorithms of the COMP-EINT construction are detailed in Figure 10. Here L_c is the byte length of the largest plaintext size for which the ciphertext is of length at most the file block length less the size of a hash function output. For example, if the block size is 4096 bytes, SHA-1 is used for hashing (whose output is 20 bytes) and 16-byte AES is used for encryption, then L_c is the largest multiple of the AES block size (i.e., 16 bytes) less than $4096 - 20 = 4076$ bytes. The value of L_c in this case is 4064 bytes.

- In the $\mathsf{Update}(i, b)$ algorithm for file F, the write counter for block i is incremented and the counter authentication information from TS_F is updated with the algorithm auth_ctr. Then block content b is compressed to b^c. If the length of b^c (denoted $|b^c|$) is at most L_c, then there

```
$F$.Update$(i, b)$ :
  $k \leftarrow F$.enc_key
  $F$.update_ctr$(i)$
  $F$.auth_ctr$()$
  $b^c \leftarrow$ compress$(b)$
  if $|b^c| \leq L_c$
    if $i \in$ RArr$_F$
      RTree$_F$.DelOffsetTree(TS$_F$, RArr$_F$, $i$)
    $c \leftarrow F.\mathsf{E}_k^{\mathsf{len}}(i, \mathsf{pad}(b^c))$
    $F$.WriteBlock$(i, c||h(i||F.\mathsf{get\_ctr}(i)||b))$
  else
    if $i \in$ RArr$_F$
      $j \leftarrow$ RArr$_F$.SearchOffset$(i)$
      RTree$_F$.UpdateTree(TS$_F$, $j$, $h(i||F.\mathsf{get\_ctr}(i)||b)$)
    else
      RTree$_F$.AppendTree(TS$_F$, $h(i||F.\mathsf{get\_ctr}(i)||b)$)
      append $i$ at end of RArr$_F$
    $c \leftarrow F.\mathsf{E}_k^{\mathsf{len}}(i, b)$
    $F$.WriteBlock$(i, c)$
```

```
$F$.Check$(i)$:
  $k \leftarrow F$.enc_key
  if $F$.check_ctr$()$ = false
    return false
  $\bar{B}_i \leftarrow F$.ReadBlock$(i)$
  if $i \in$ RArr$_F$
    $B_i \leftarrow F.\mathsf{D}_k^{\mathsf{len}}(i, \bar{B}_i)$
    $j \leftarrow$ RArr$_F$.SearchOffset$(i)$
    return RTree$_F$.CheckTree(
          TS$_F$, $j$, $h(i||F.\mathsf{get\_ctr}(i)||B_i)$)
  else
    parse $\bar{B}_i$ as $c||hval$
    $b^c \leftarrow$ unpad$(F.\mathsf{D}_k^{\mathsf{len}}(i, c))$
    $b \leftarrow$ decompress$(b^c)$
    if $hval = h(i||F.\mathsf{get\_ctr}(i)||b)$
      return true
    else
      return false
```

```
$F$.Append$(b)$ :
  $k \leftarrow F$.enc_key
  $F$.update_ctr$(i)$
  $F$.auth_ctr$()$
  $n \leftarrow F$.blocks
  $b^c \leftarrow$ compress$(b)$
  if $|b^c| \leq L_c$
    $c \leftarrow F.\mathsf{E}_k^{\mathsf{len}}(n+1, \mathsf{pad}(b^c))$
    $F$.WriteBlock$(i, c||h(n+1||F.\mathsf{get\_ctr}(n+1)||b))$
  else
    RTree$_F$.AppendTree(TS$_F$, $h(n+1||F.\mathsf{get\_ctr}(n+1)||b)$)
    append $n+1$ at end of RArr$_F$
    $c \leftarrow F.\mathsf{E}_k^{\mathsf{len}}(n+1, b)$
    $F$.WriteBlock$(n+1, c)$
```

```
$F$.Delete$()$:
  $F$.del_ctr$(n)$
  $F$.auth_ctr$()$
  $n \leftarrow F$.blocks
  if $n \in$ RArr$_F$
    RTree$_F$.DelOffsetTree(TS$_F$, RArr$_F$, $n$)
  delete $B_n$ from file $F$
```

Figure 10: The Update, Check, Append and Delete algorithms for the COMP-EINT construction.

is room to store the hash of the block content inside the block. In this case, the hash of the previous block content stored at the same address is deleted from the Merkle tree RTree$_F$, if necessary. The compressed block is padded and encrypted, and then stored with its hash in the i-th block of F. Otherwise, if the block cannot be compressed enough, then its hash has to be inserted into the Merkle tree RTree$_F$. The block content b is then encrypted with a length-preserving stateful encryption scheme using the key for the file and is stored in the i-th block of F.

- In the Check(i) algorithm for file F, the authentication information from TS$_F$ for the block counters is checked first. There are two cases to consider. First, if the hash of the block content stored at the i-th block of F is authenticated through the Merkle tree RTree$_F$, then the block is decrypted and algorithm CheckTree is called. Otherwise, the hash of the block content is stored at the end of the block and we can thus parse the i-th block of F as $c||hval$.

c has to be decrypted, unpadded and decompressed, in order to obtain the original block content b. The hash value $hval$ stored in the block is checked to match the hash of the block index i concatenated with the write counter for block i and block content b.

- To append a new block b to file F with n blocks using the $\mathsf{Append}(b)$ algorithm, the write counter for the new block is initialized to 1 and the authentication information for counters stored in TS_F is updated. Block b is then compressed. If it has an adequate compression level, then the compressed block is padded and encrypted, and a hash is concatenated at the end of the new block. Otherwise, a new hash is appended to the Merkle tree RTree_F and an encryption of b is stored in the $(n+1)$-th block of F.

- To delete the last block from file F with n blocks, algorithm Delete is used. The write counter for the block is deleted and the authentication information for counters is updated. If the n-th block is authenticated through RTree_F, then its hash has to be removed from the tree by calling the algorithm DelOffsetTree.

The construction COMP-EINT prevents against replay attacks by hashing write counters for file blocks together with block contents and authenticating the write counters in trusted storage. It meets all the security properties of RAND-EINT and MT-EINT.

6 Implementation

Our integrity algorithms are very general and they can be integrated into any cryptographic file system in either the kernel or userspace. For the purpose of evaluating and comparing their performance, we implemented them in EncFS [10], an open-source user-level file system that transparently encrypts file blocks. EncFS uses the FUSE [13] library to provide the file system interface. FUSE provides a simple library API for implementing file systems and it has been integrated into recent versions of the Linux kernel.

In EncFS, files are divided into fixed-size blocks and each block is encrypted individually. Several ciphers such as AES and Blowfish in CBC mode are available for block encryption. We implemented in EncFS the three constructions that provide integrity: MT-EINT, RAND-EINT and COMP-EINT. While any length-preseving encryption scheme can be used in the MT-EINT and COMP-EINT constructions, RAND-EINT is constrained to use a tweakable cipher for encrypting file blocks. We choose to encrypt file blocks in MT-EINT and COMP-EINT with the length-preserving stateful encryption derived from the AES cipher in CBC mode, and use the CMC tweakable cipher [15] as the encryption method in RAND-EINT. For compressing and decompressing blocks in COMP-EINT we used the zlib library [29].

Our prototype architecture is depicted in Figure 11. We only modified the client-side of EncFS to include the CMC cipher for block encryption and the new integrity algorithms. The server could use any underlying file system for the storage of the encrypted files. The Merkle trees for integrity RTree_F and the index arrays of the random-looking blocks RArr_F are stored with the encrypted files in the untrusted storage space on the server. For faster integrity checking (in particular to improve the running time of the SearchOffset algorithm used in the Update and Check algorithms of the $\mathsf{RAND\text{-}EINT}$ and $\mathsf{COMP\text{-}EINT}$ constructions), we also keep the array RArr_F for each file, ordered by indices, in untrusted storage. The roots of the trees RTree_F, and the arrays $\mathsf{IntStart}_F$ and CtrVal_F or their hashes (if they are too large) are stored in a trusted storage space, possibly on a dedicated meta-data server. In our current implementation, we use two extended attributes for each file F, one for the root of RTree_F and the second for the arrays $\mathsf{IntStart}_F$ and CtrVal_F, or their hashes (see the counter authentication algorithm from Section 4.2).

By default, EncFS caches the last block content written or read from the disk. In our implementation, we cached the last arrays RArr_F, $\mathsf{IntStart}_F$ and CtrVal_F used in a block update or check operation. Since these arrays are typically small (a few hundred bytes), they easily fit into memory. We also evaluate the effect of caching of Merkle trees in our system.

7 Performance Evaluation

In this section, we evaluate the performance of the new integrity constructions for encrypted storage $\mathsf{RAND\text{-}EINT}$ and $\mathsf{COMP\text{-}EINT}$ compared to that of $\mathsf{MT\text{-}EINT}$. We ran our experiments on a 2.8 GHz Intel D processor machine with 1GB of RAM, running SuSE Linux 9.3 with kernel version 2.6.11. The hard disk used was an 80GB SATA 7200 RPM Maxtor.

The main challenge we faced in evaluating the proposed constructions was to come up with representative file system workloads. While the performance of the Merkle tree construction is predictable independently of the workload, the performance of the new integrity algorithms is highly dependent on the file contents accessed, in particular on the randomness of block contents. To our knowledge, there are no public traces that contain file access patterns, as well as the contents of the file blocks read and written. Due to the privacy implications of releasing actual users' data, we expect it to be nearly impossible to get such traces from a widely used system. However, we have access to three public NFS Harvard traces [9] that contain NFS traffic from several of Harvard's campus servers. The traces were collected at the level of individual NFS operations and for each read and write operation they contain information about the file identifier, the accessed offset in the file and the size of the request.

To evaluate the integrity algorithms proposed in this paper, we perform two sets of experiments. In the first one, we strive to demonstrate how the performance of the new constructions varies for different file contents. For that, we use representative files from a Linux distribution installed on one of our desktop machines, together with other files from the user's home directory, divided into several file types. We identify five file types of interest: text, object, executables, images, and compressed files, and divide the collected files according to these five classes. All files of a particular type are first encrypted and the integrity information for them is built; then they are decrypted and checked for integrity. We report the performance results for the files with the majority of blocks not random-looking (i.e., text, executable and object) and for those with mostly random-looking blocks (i.e., image and compressed). In this experiment, all files are written and read sequentially, and as such the access pattern is not a realistic one.

In the second set of experiments, we evaluate the effect of more realistic access patterns on the performance of the integrity schemes, using the NFS Harvard traces. As the Harvard traces do not contain information about actual file block contents written to the disks, we generate synthetic

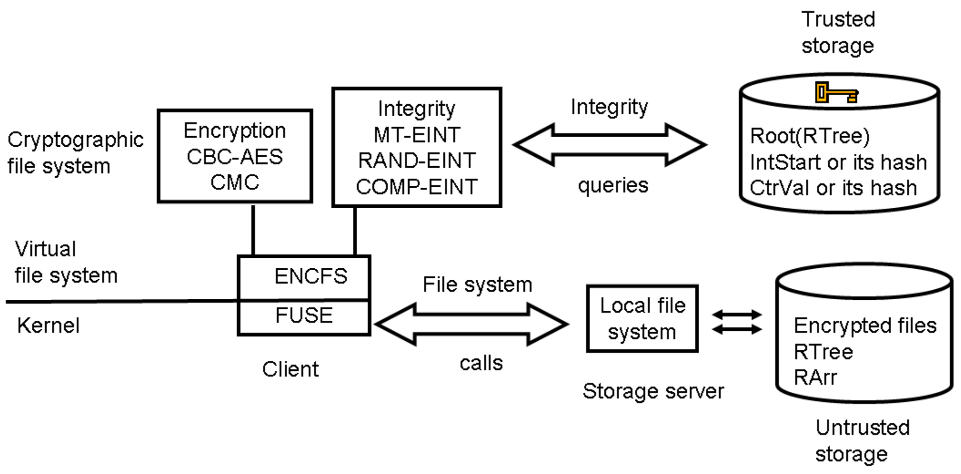

Figure 11: Prototype architecture.

block contents for each block write request. We define two types of block contents: low-entropy and high-entropy, and perform experiments assuming that all block contents are either low or high entropy. These extreme workloads represent the "best" and "worst"-case for the new algorithms, respectively. We also consider a "middle"-case, in which a block is random-looking with a 50% probability and plot the performance results of the new schemes relative to the Merkle tree integrity algorithm for the best, middle and worst cases.

To evaluate exactly the performance of the new constructions, it would be beneficial to know what is the percentage of random-looking blocks in practical workloads. There exists only one previous study [24] of which we are aware to provide statistics on this, but only for one desktop computer. In this study, only about two percent of the blocks written to disk were random-looking. As our results show, for traces with less than half of the blocks random-looking, the new schemes improve on the Merkle tree construction with respect to all the three metrics discussed in Section 2.

Finally, we also evaluate how the amount of trusted storage per file affects the storage of the counter intervals for replay detection in the three NFS Harvard traces.

7.1 The Impact of File Block Contents on Integrity Performance

File traces. For this set of experiments, we build five file traces collected from one of our desktop machines: (1) *text files* are files with extensions .txt, .tex, .c, .h, .cpp, .java, .ps, .pdf; (2) *object files* are system library files from the directory /usr/local/lib; (3) *executable files* are system executable files from directory /usr/local/bin; (4) *image files* are JPEG files and (5) *compressed files* are gzipped tar archives. Several characteristics of each trace, including the total trace size, the number of files in each trace, the minimum, average and maximum file sizes are given in Table 2. Table 3 shows the fraction of file blocks that are considered random-looking by the entropy test.

Cryptographic file systems. We consider five cryptographic file systems: (1)"AES - no integrity" encrypts file blocks with AES in CBC mode and no integrity is provided; (2)"CMC -

	Total size	No files	Min file size	Max file size	Avg file size
Text	245 MB	808	27 bytes	34.94 MB	307.11 KB
Objects	217 MB	28	15 bytes	92.66 MB	7.71 MB
Executables	341 MB	3029	24 bytes	13.21 MB	112.84 KB
Image	189 MB	641	17 bytes	2.24 MB	198.4 KB
Compressed	249 MB	2	80.44 MB	167.65 MB	124.05 MB

Table 2: File trace characteristics.

Block size	128 bytes	256 bytes	512 bytes	1024 bytes	2048 bytes	4096 bytes
Text	0.1075	0.087	0.0652	0.0524	0.0409	0.0351
Object	0.0003	0.0002	0.0002	0.0002	0.0002	0.0001
Executables	0.003	0.0019	0.0015	0.0013	0.0011	0.0009
Image	0.9421	0.9277	0.8794	0.788	0.6176	0.502
Compressed	0.936	0.8936	0.8481	0.8292	0.7915	0.7812

Table 3: Fraction of random-looking blocks in the five file traces.

no integrity" encrypts file blocks with CMC and no integrity is provided; (3)"AES-Merkle" encrypts file blocks with AES in CBC mode and MT-EINT is used for integrity; (4) "CMC-Entropy" encrypts file blocks with CMC and RAND-EINT test is used for integrity; (5)"AES-Compression" encrypts file blocks with AES in CBC mode and COMP-EINT is used for integrity.

Experiments. For each cryptographic file system, we first write the files from each trace; this has the effect of automatically encrypting the files, and running the Update or Append algorithm of the integrity method for each file block. Second, we read all files from each trace; this has the effect of automatically decrypting the files, and running the Check algorithm of the integrity method for each file block.

We plot the total file write and read time in seconds for the five cryptographic file systems as a function of different block sizes. We also plot the average integrity bandwidth per block. Finally, we plot the cumulative size of the untrusted storage US_F for all files from each trace. All the graphs are done in a log-log scale, as all the three metrics considered decrease exponentially for block sizes varying from 128 to 4096 bytes. We show the combined graphs for low-entropy files (text, object and executable files) in Figure 12 and for high-entropy files (compressed and image files) in Figure 13. We also present in Tables 4 and 5 the numerical values for all metrics of interest for all five file traces in the experiment with 4096-byte blocks.

	Write Text	Write Obj	Write Exe	Write Image	Write Comp	Read Text	Read Obj	Read Exe	Read Image	Read Comp
AES - no integrity	20.18	13.75	54.85	8.81	18.57	9.07	7.93	26.26	7.52	8.63
CMC - no integrity	26.94	20.13	65.6	14.97	23.31	17.92	15.79	34.1	13.84	18.14
AES-Merkle	53.67	43.08	99.43	35.86	50.46	40.55	45.76	66.96	27.88	55.82
CMC-Entropy	31.78	22.33	73.25	37.56	58.81	24.71	21.19	49.34	30.44	62.44
AES-Compression	51.32	43.54	105.83	59.88	78.82	18.49	15.44	41.73	27.31	51.93

Table 4: Time (in s) to write and read file traces for 4096-bytes blocks.

	Avg Bw Text	Avg Bw Obj	Avg Bw Exe	Avg Bw Image	Avg Bw Comp	Int Text	Int Obj	Int Exe	Int Image	Int Comp
AES-Merkle	132.86	204.38	111.22	106.42	227.09	8.1	2.34	26.26	6.1	2.49
CMC-Entropy	4.71	0.16	0.52	60.74	211.08	3.55	0.13	12.24	5.38	2.06
AES-Compression	9.93	3.01	3.66	113.26	229.44	6.55	0.22	24.38	6.04	2.22

Table 5: Average integrity bandwidth per block (in bytes) and total untrusted storage for integrity (in MB) for file traces for 4096-bytes blocks.

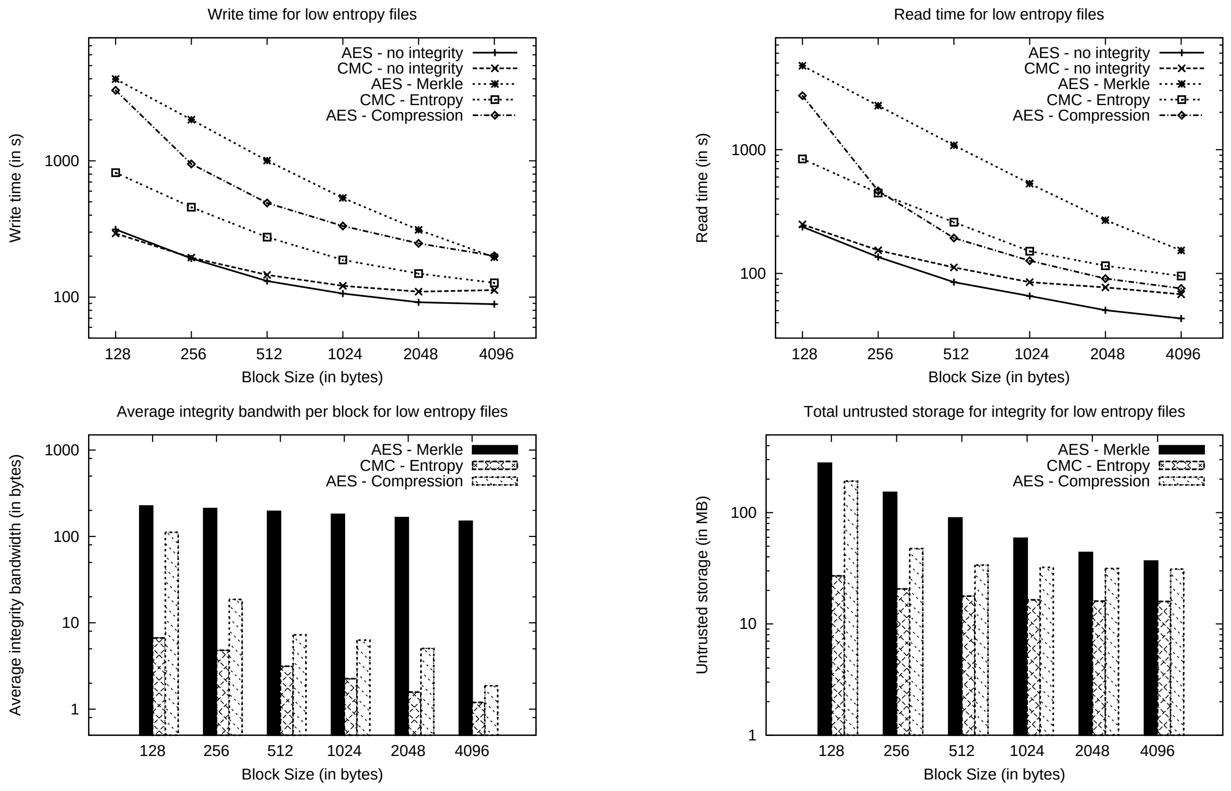

Figure 12: Evaluation for Low-Entropy Files (Text, Object and Executable Files)

Results for low-entropy files. For traces with a low percent of random-looking blocks (text, object and executable files), CMC-Entropy outperforms AES-Merkle with respect to all the metrics considered. For 4096-byte blocks, the performance of CMC-Entropy compared to that of AES-Merkle is improved by 35.08% for writes and 37.86% for reads. The performance of the AES-Compression file system is very different in the write and read experiments. For 4096-byte blocks, the write time of AES-Compression is within 2% of the write time of AES-Merkle. However, in the read experiment, AES-Compression outperforms AES-Merkle by 50.63%. The reason for this discrepancy between the read and write performance is the low cost of the decompression algorithm used for reading file blocks compared to the compression algorithm used for writing file blocks in AES-Compression. We further break down the write and read performance costs of AES-Merkle, CMC-Entropy and AES-Compression into components in the micro-benchmarks paragraph later this section.

The integrity bandwidth of CMC-Entropy and AES-Compression is 84.08 and 26.49 times, respectively, lower than that of AES-Merkle. The untrusted storage for integrity for CMC-Entropy and AES-Compression is reduced 2.3 and 1.17 times, respectively, compared to AES-Merkle.

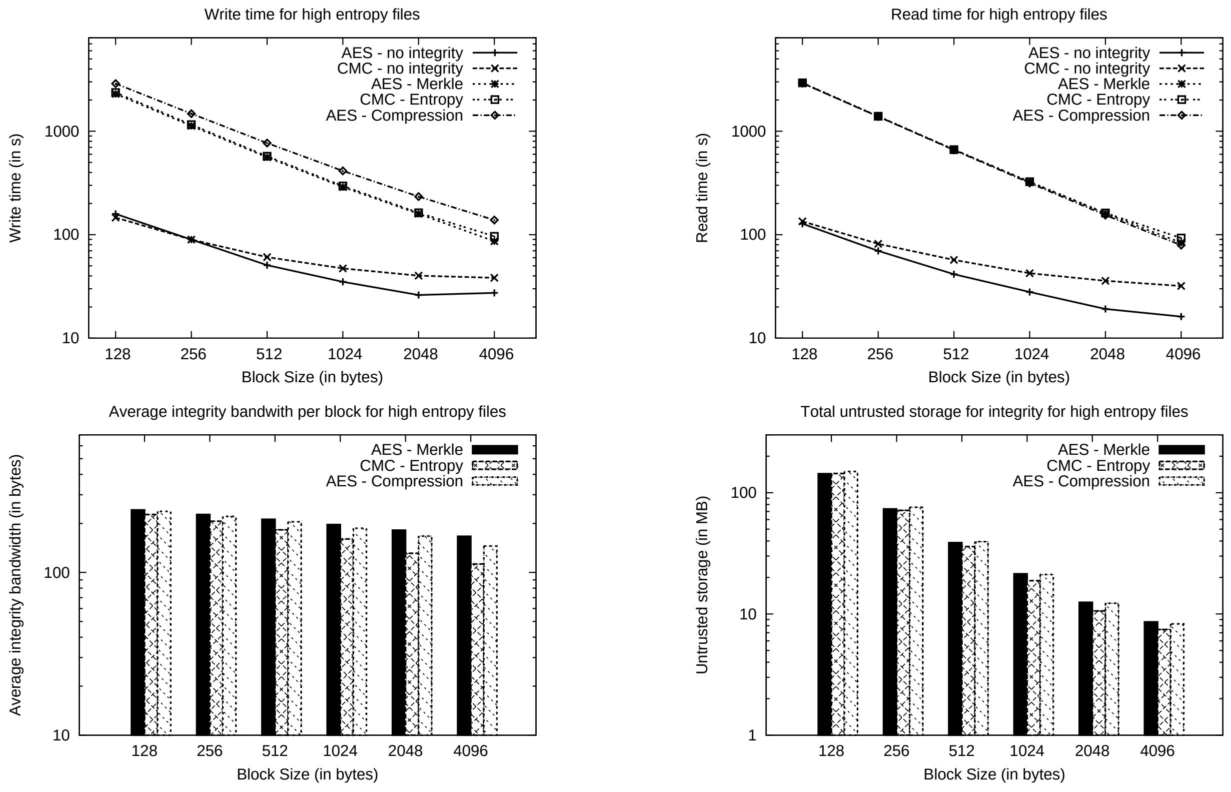

Figure 13: Evaluation for High-Entropy Files (Image and Compressed Files)

Results for high-entropy files. For traces with a high percent of random-looking blocks (image and compressed files), CMC-Entropy adds a maximum performance overhead of 11.64% for writes and 10.96% for reads compared to AES-Merkle. The performance of AES-Compression is very different in the write and read experiments due to the cost difference of compression and decompression. AES-Compression adds a write performance overhead of 60.68% compared to AES-Merkle, and improves the read performance of AES-Merkle by 5.32%.

For 4096-byte blocks, the average integrity bandwidth needed by CMC-Entropy and AES-Compression is lower by 16.45% and higher by 2.44%, respectively, than that used by AES-Merkle. The untrusted storage for integrity used by CMC-Entropy and AES-Compression compared to that of AES-Merkle is improved by 13.38% and 3.84%, respectively, for 4096-byte blocks.

Micro-benchmarks. To better understand our results, we present a micro-benchmark evaluation for 4096-byte blocks for text and compressed files in Figure 14. We separate the total time incurred by the write and read experiments into the following components: encryption/decryption time

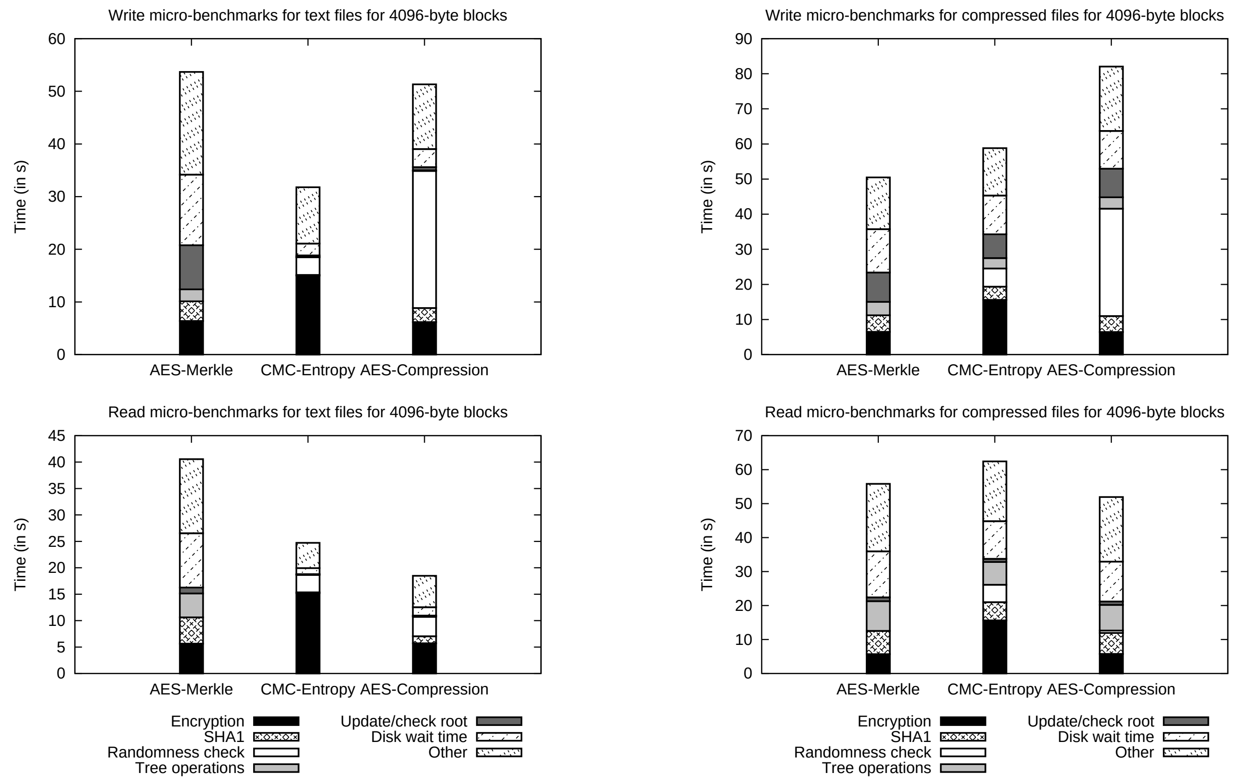

Figure 14: Micro-benchmarks for text and compressed files.

(either AES or CMC); SHA1 hashing time; randomness check time (either the entropy test for CMC-Entropy or compression/decompression time for AES-Compression); Merkle tree operations (e.g., given a leaf index, find its index in inorder traversal or given an inorder index of a node in the tree, find the inorder index of its sibling and parent); the time to update and check the root of the tree (the root of the Merkle tree is stored as an extended attribute for the file) and disk waiting time.

The cost of CMC encryption and decryption is about 2.5 times higher than that of AES encryption and decryption in CBC mode. Decompression is between 4 and 6 times faster than compression and this accounts for the good read performance of AES-Compression compared to both AES-Merkle and CMC-Entropy.

A substantial amount of the AES-Merkle overhead is due to disk waiting time (25%) and the time to update and check the root of the Merkle tree (15%). In contrast, due to smaller sizes of the Merkle trees in the CMC-Entropy and AES-Compression file systems, the disk waiting time and the time to update and check the root of the tree for text files are only 7% and 0.9% of the total write time for CMC-Entropy and 6% and 1%, respectively, for AES-Compression. For compressed files, the disk waiting time and the time to update and check the root of the tree represent 17% and 11% of the total write time for CMC-Entropy and 14% and 10%, respectively, for AES-Compression.

The results suggests that caching of Merkle trees in the file system might reduce the disk waiting time and improve the performance of all three integrity constructions, and specifically that of the AES-Merkle algorithm. We present our results on caching next.

Caching of Merkle trees. We have implemented in our system a cache that stores the hashes read from Merkle trees. The cache uses the least-recent used (i.e., LRU) replacement policy. We have experimented with two algorithms that have different tradeoffs. The first one implements the cache using a double-linked list, in which the most recently used block is moved in front of the list. The algorithm optimizes the time to search for the least-recent used block, at the expense of growing the search time linearly with the number of items in the cache. The second algorithm implements the cache as a hash table indexed by the (file name, block index in file) tuple. To determine the least recently used block, the algorithm stores a global counter that is incremented every time an element is inserted in (or used from) the cache, as well as a counter for each block in the cache that denotes the value of the global counter when that block has been most recently used. In this algorithm, the search of a block involves one table lookup and a search in the short list of items that hash to the same value (in our experiments, the largest bucket of items that hashed to the same value contained three items). On the other hand, the time to replace an item in the cache is linear in the number of items in the cache.

We performed experiments with both algorithms by varying the size of the cache between 0 and 4KB. We found out that for caches smaller than 4KB, the double-linked list algorithm outperforms the hash-table algorithm, but as the size of the cache grows the hash-table algorithm becomes faster. It turns out that the cache hit rate becomes very high for small caches (e.g., for a 512-byte cache, the cache hit rate is 77% for text files and 86% for compressed files) and the optimum value for the size of the cache is 512 or 1024 bytes. The high cache hit rate is explained by the sequentiality of file accesses in the experiment that we performed.

We represent the write and read performance for both low and high entropy files for the faster double-linked list algorithm in Figure 15. The performance of CMC-Entropy at read is improved by only 2% for low-entropy files, while the performance of AES-Merkle is improved by 7%. On the other hand, for high-entropy files, all three constructions are improved by at least 7% and at most 11%. We do not obtain a higher benefit from caching in our system because in our initial experiments there is an implicit caching mechanism of the file system (as the Merkle trees are stored in the file system).

7.2 The Impact of File Access Patterns on Integrity Performance

File traces. We considered a subset of the three NFS Harvard traces [9] (LAIR, DEASNA and HOME02), each collected during one day. We show several characteristics of each trace, including the number of files and the total number of block write and read operations, in Table 6. The block size in these traces is 4096 bytes.

Experiments. We replayed each of the three traces with three types of block contents: all low-entropy, all high-entropy and 50% high-entropy. For each experiment, we measured the total running time, the average integrity bandwidth and the total untrusted storage for integrity for CMC-Entropy and AES-Compression relative to AES-Merkle and plot the results in Figure 16.

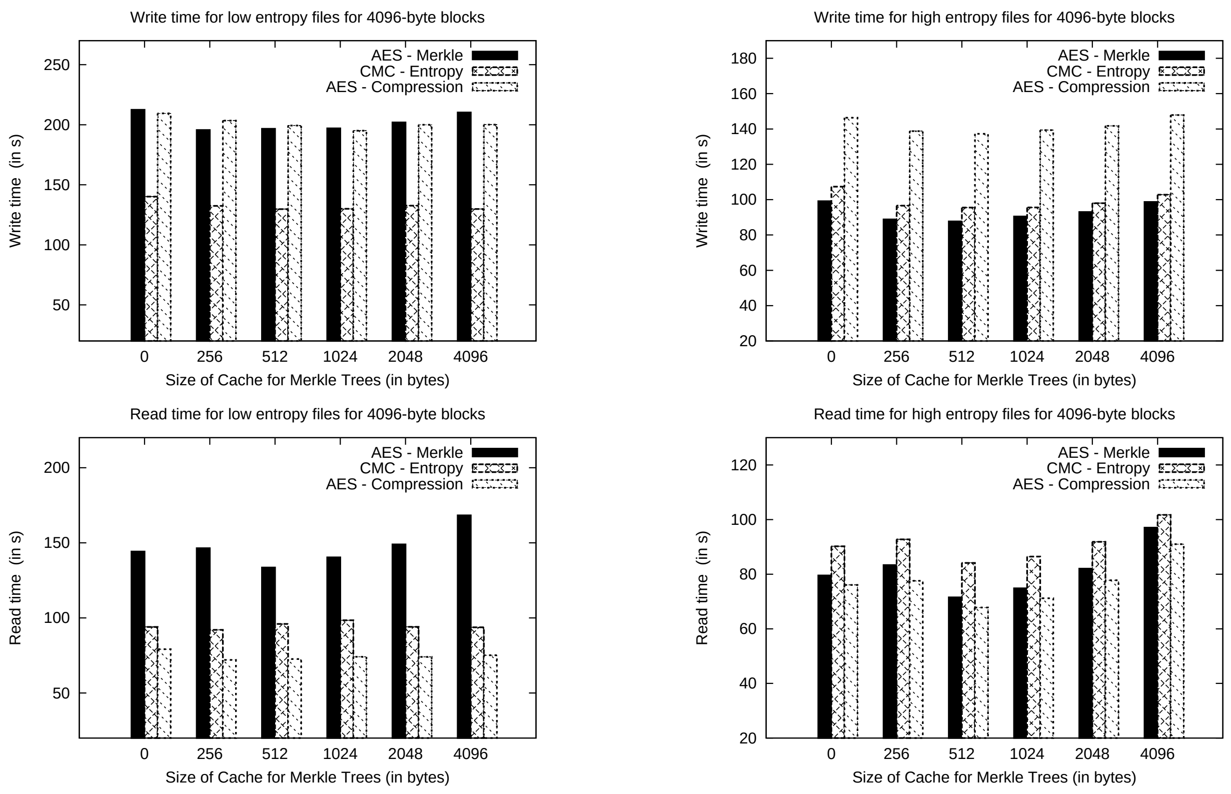

Figure 15: Write and read time for low and high-entropy files as a function of the size of cache.

	Number of files	Number writes	Number reads
LAIR	7017	66331	23281
DEASNA	890	64091	521
HOME02	183	89425	11815

Table 6: NFS Harvard trace characteristics.

We represent the performance of AES-Merkle as the horizontal axis in these graphs and the performance of CMC-Entropy and AES-Compression as a percentage relative to AES-Merkle. The points above the horizontal axis are percentage overheads compared to AES-Merkle, and the points below the horizontal axis represent percentage improvements relative to AES-Merkle. The labels on the graphs denote the percent of random-looking blocks synthetically generated.

Results for low-entropy blocks. For low-entropy block contents, the performance improvements of CMC-Entropy compared to AES-Merkle are 29.11% for LAIR, 46.33% for DEASNA and 53.36% for HOME02. The performance of AES-Compression improves when the ratio of read to write operations increases. AES-Compression performs best for the LAIR trace, in which it improves upon CMC-Entropy by 13.14%. For the DEASNA trace, AES-Compression adds an overhead of 9.98% compared to CMC-Entropy, and for the HOME02 trace the performance of CMC-Entropy and AES-Compression are very close.

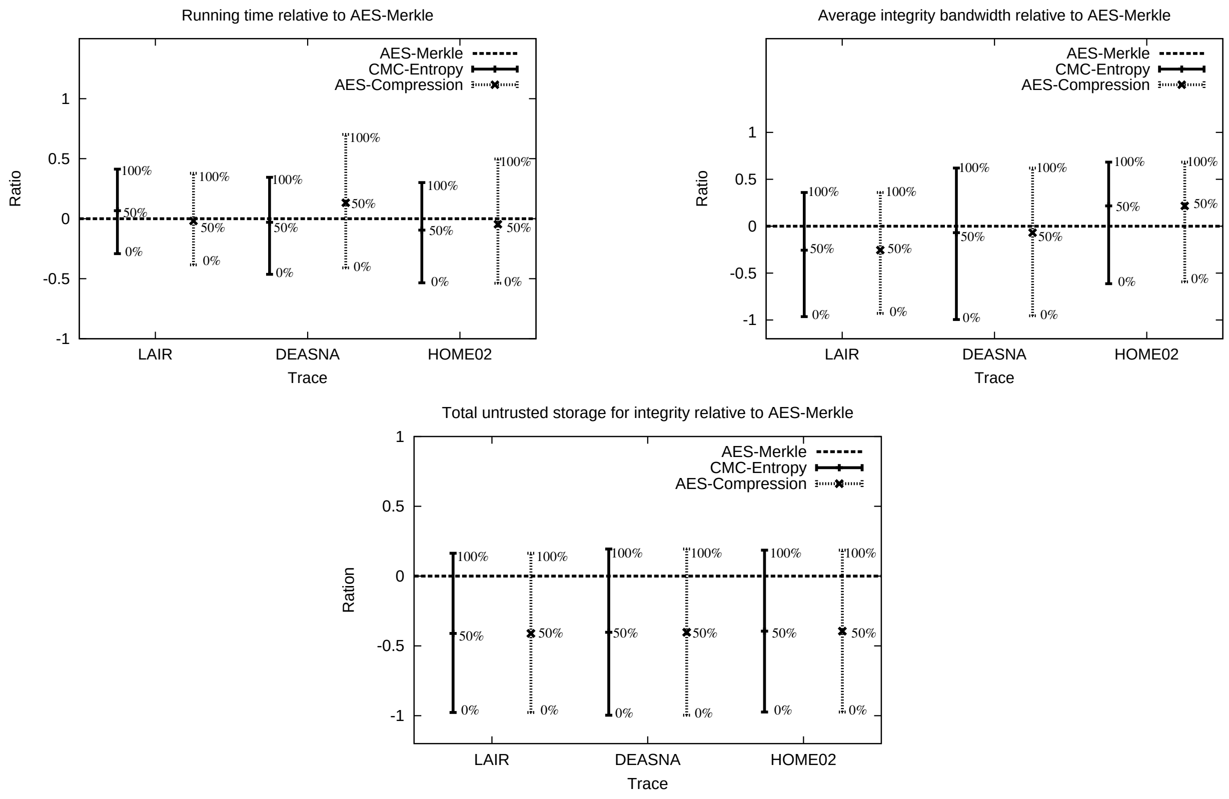

Figure 16: Running time, average integrity bandwidth and storage for integrity of CMC-Entropy and AES-Compression relative to AES-Merkle. Labels on the graphs represent percentage of random-looking blocks.

The average integrity bandwidth for CMC-Entropy is lower than the bandwidth used by AES-Merkle by a factor of 28.04 for LAIR, 238.82 for DEASNA and 2.58 for HOME02. The average integrity bandwidth for AES-Compression is lower than the bandwidth used by AES-Merkle by a factor of 14.29 for LAIR, 22.2 for DEASNA and 2.45 for HOME02. Similarly, the untrusted storage for integrity for CMC-Entropy and AES-Compression improves that used by AES-Merkle by a factor of 44.31 for LAIR, 365.71 for DEASNA and 37.42 for HOME02.

Results for high-entropy blocks. For high-entropy blocks, CMC-Entropy adds a performance overhead compared to AES-Merkle of at most 41.25% for LAIR, 34.49% for DEASNA and 30.15% for HOME02. On the other hand, the performance overhead added by AES-Compression compared to AES-Merkle is at most 37.8% for LAIR, 70.18% for DEASNA and 49.76% for HOME02.

The bandwidth overhead of CMC-Entropy and AES-Compression is at most 35.99% for LAIR, 61.99% for DEASNA and 68.32% for HOME02. The storage overhead of CMC-Entropy and AES-Compression compared to AES-Merkle is between 16% and 19% for all the three traces.

Results for 50% low-entropy blocks. The performance of CMC-Entropy and AES-Compression are within 9% and 13%, respectively, of that of AES-Merkle for traces with 50% high-entropy blocks. The average integrity bandwidth of both CMC-Entropy and AES-Compression is within 25% of that of AES-Merkle and the storage for integrity of both CMC-Entropy and AES-Compression is 40% lower than that of AES-Merkle.

7.3 Amount of Trusted Storage

Finally, we perform some experiments to evaluate how the storage of the counter intervals, in particular the arrays $\mathsf{IntStart}_F$ and CtrVal_F, is affected by the amount of trusted storage per file. For that, we plot the number of files for which we need to keep the arrays $\mathsf{IntStart}_F$ and CtrVal_F in the untrusted storage space, as a function of the amount of trusted storage per file. The results for the three traces are in Figure 17. We conclude that a value of 200 bytes of constant storage per file (which we have used in our experiments) is enough to keep the counter intervals for all the files in the LAIR and DEASNA traces, and about 88% percent of the files in the HOME02 trace.

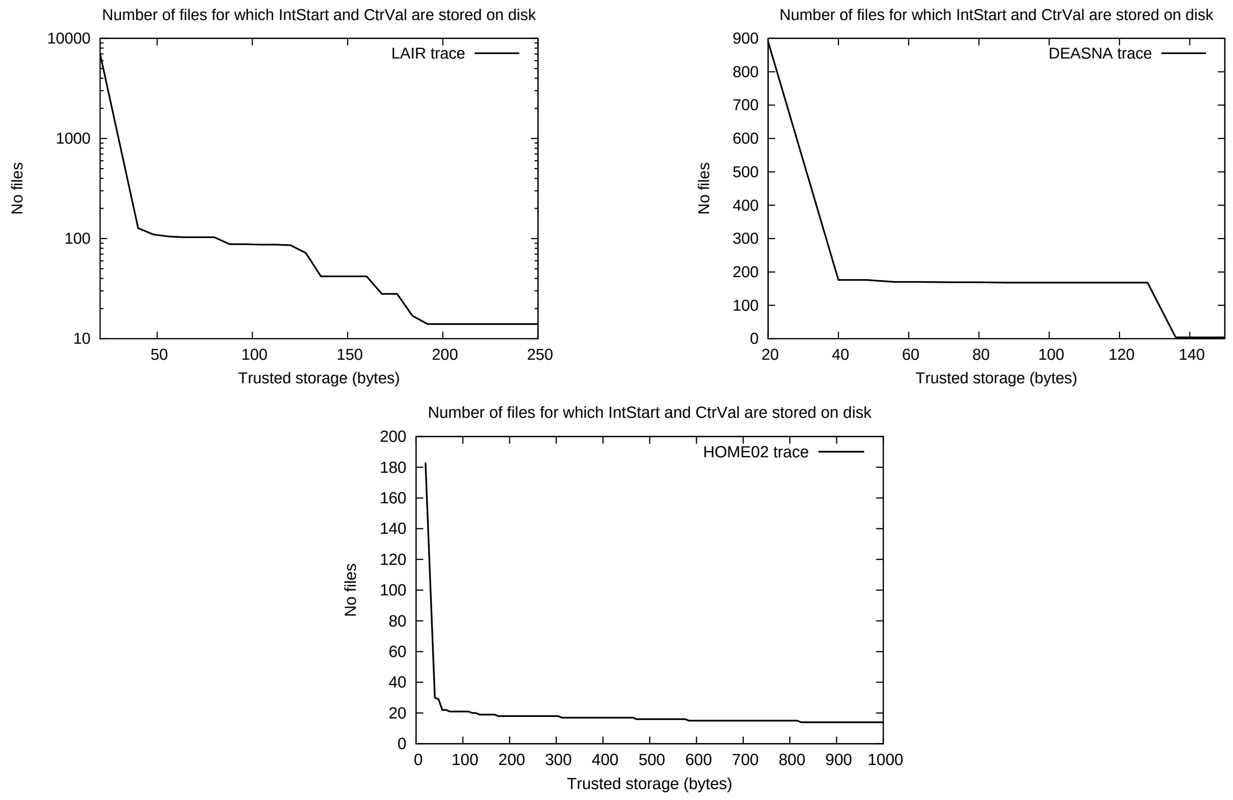

Figure 17: Number of files for which the counter intervals are stored in the untrusted storage space.

8 Discussion

From the evaluation of the three constructions, it follows that none of the schemes is a clear winner over the others. While the performance of AES-Merkle is not dependent on workloads, the performance of both CMC-Entropy and AES-Compression is greatly affected by file block contents written to disks and file access patterns. The scheme AES-Compression is the most variable with respect to performance among the three constructions and CMC-Entropy performs best in most cases with respect to the average integrity bandwidth and the untrusted storage for integrity metrics.

	Majority writes	Majority reads
Majority low-entropy contents	CMC-Entropy	AES-Compression
Majority high-entropy contents	AES-Merkle	AES-Merkle

Table 7: Choosing a construction based on block contents and access patterns.

We recommend that all three constructions be implemented in a cryptographic file system. An application can choose the best scheme based on its typical workload. We offer several guidelines for choosing the best suited integrity scheme for encrypted storage in Table 7.

The new algorithms that we propose can be applied in other settings in which authentication of data stored on untrusted storage is desired. One example is checking the integrity of arbitrarily-large memory in a secure processor using only a constant amount of trusted storage [4, 7]. In this setting, a trusted checker maintains a constant amount of trusted storage and, possibly, a cache of data blocks most recently read from the main memory. The goal is for the checker to verify the integrity of the untrusted memory using a small bandwidth overhead.

The algorithms described in this chapter can be only used in applications where the data that needs to be authenticated is encrypted. However, the COMP-EINT integrity algorithm can be easily modified to fit into a setting in which data is only authenticated and not encrypted, and can thus replace Merkle trees in such applications. On the other hand, the RAND-EINT integrity algorithm is only suitable in a setting in which data is encrypted with a tweakable cipher, as the integrity guarantees of this algorithm are based on the security properties of such ciphers.

In checking the integrity of untrused memory, the memory is divided into smaller blocks (e.g., 64 or 128 bytes) than in a cryptographic file system, where the typical granularity of securing file blocks is 1KB or 4KB. In fact, we chose the block values between 128 bytes and 4KB in our experiments to show the applicability of our integrity algorithms to other applications besides cryptographic file systems.

9 Related Work

As discussed in Section 1, it is possible to reduce the trusted storage costs for many files to that of only one by writing the trusted storage for multiple files to a cryptographically protected file (stored in untrusted storage). An alternative that is employed in numerous systems (e.g., SFSRO [12], Cepheus [11], FARSITE [1], Plutus [17], and SUNDR [19]) is to digitally sign the trusted storage for a file—in those systems, the root of a Merkle tree for the file—and write this digitally signed value to untrusted storage. This technique similarly reduces the trusted storage for many files to a constant-size trusted value, in this case the trusted public key for verifying the signatures. The techniques we propose here can replace Merkle trees equally well in this context.

Still other integrity mechanisms exist for cryptographic file systems. For example, a common integrity method, used in systems such as TCFS [5], PFS [28] and SNAD [23], is to store a hash or message authentication code (MAC) for each file block. However, this results in trusted (in the case of hashes) or untrusted (in the case of MACs) integrity storage for every block, which we avoid in our constructions (except for workloads consisting of all random files). Tripwire [18] is a user-level tool that computes a hash per file and stores it in trusted storage. While this approach achieves constant trusted storage for integrity per file, the entire file must be read to verify the integrity of any block, violating one of our design goals (see Section 1). The same is true for SiRiUS [14], which applies a digital signature to each full file.

Riedel et al. [25] provide a framework for extensively evaluating the security of existing storage systems. A recent survey of the integrity methods used in different storage system is by Sivathanu et al. [27]. We refer the readers to these surveys for more discussion of secure storage systems.

Another area related to our work is that of checking the integrity of arbitrarily large untrusted memory using only a small, constant-size trusted memory. A first solution to this problem by Blum et al. [4] is to check the integrity of each memory operation using a Merkle tree. This results in a logarithmic bandwidth overhead in the total number of memory blocks. Recent solutions try to reduce the logarithmic bandwidth to almost a constant value. The main idea by Clarke et al. [7] is to verify the integrity of the memory only when a *critical operation* is encountered. The integrity of sequences of operations between critical operations is checked by aggregating these operations in a log using incremental hash functions [6]. In contrast, in our model we need to be able to check the integrity of *each* file block read from the storage servers.

10 Conclusion

We have proposed two new integrity constructions, RAND-EINT and COMP-EINT, for encrypted file systems. Our constructions exploit the typical low entropy of block contents and sequentiality of file block writes. We have evaluated the performance of the new constructions relative to the widely used Merkle tree integrity algorithm, using file traces from a standard Linux distribution and NFS traces collected at the Harvard university. Our experimental evaluation demonstrates that each construction is best suited for particular workloads.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, Usenix, 2002.

[2] "Advanced encryption standard." Federal Information Processing Standards Publication 197, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, Nov. 2001.

[3] J. Black and H. Urtubia, "Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption," in *Proc. 11th USENIX Security Symposium*, pp. 327–338, 2002.

[4] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica*, vol. 12, pp. 225–244, 1994.

[5] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for Unix," in *Proc. USENIX Annual Technical Conference 2001, Freenix Track*, pp. 199–212, 2001.

[6] D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Proc. Asiacrypt 2003*, vol. 2894 of *Lecture Notes in Computer Science*, pp. 188–207, Springer-Verlag, 2003.

[7] D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas, "Towards constant bandwidth overhead integrity checking of untrusted data," in *Proc. 26th IEEE Symposium on Security and Privacy*, pp. 139–153, 2005.

[8] "DES modes of operation." Federal Information Processing Standards Publication 81, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1980.

[9] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive nfs tracing of email and research workloads," in *Proc. Second USENIX Conference on File and Storage Technologies (FAST)*, pp. 203–216, 2003.

[10] "EncFS encrypted filesystem." `http://arg0.net/wiki/encfs`.

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

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

[13] "FUSE: filesystem in userspace." `http://fuse.sourceforge.net`.

[14] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pp. 131–145, ISOC, 2003.

[15] S. Halevi and P. Rogaway, "A tweakable enciphering mode," in *Proc. Crypto 2003*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 482–499, Springer-Verlag, 2003.

[16] S. Halevi and P. Rogaway, "A parallelizable enciphering mode," in *Proc. The RSA conference - Cryptographer's track (RSA-CT)*, vol. 2964 of *Lecture Notes in Computer Science*, pp. 292–304, Springer-Verlag, 2004.

[17] 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.

[18] G. H. Kim and E. H. Spafford, "The design and implementation of Tripwire: A filesystem integrity checker," in *Proc. Second ACM Conference on Computer and Communication Security (CCS)*, pp. 18–29, 1994.

[19] J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository," in *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pp. 121–136, Usenix, 2004.

[20] M. Liskov, R. Rivest, and D. Wagner, "Tweakable block ciphers," in *Proc. Crypto 2002*, vol. 2442 of *Lecture Notes in Computer Science*, pp. 31–46, Springer-Verlag, 2002.

[21] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank - a heterogeneous scalable SAN file system," *IBM Systems Journal*, vol. 42, no. 2, 2003.

[22] R. Merkle, "A certified digital signature," in *Proc. Crypto 1989*, vol. 435 of *Lecture Notes in Computer Science*, pp. 218–238, Springer-Verlag, 1989.

[23] E. Miller, D. Long, W. Freeman, and B. Reed, "Strong security for distributed file systems," in *Proc. the First USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[24] A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity," in *Proc. Network and Distributed System Security Symposium (NDSS)*, ISOC, 2005.

[25] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 15–30, 2002.

[26] "Secure hash standard." Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/National Institute of Standards and Technology, National Technical Information Service, Springfield, Virginia, Apr. 1995.

[27] G. Sivathanu, C. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proc. ACM Workshop on Storage Security and Survivability*, 2005.

[28] C. A. Stein, J. Howard, and M. I. Seltzer, "Unifying file system protection," in *Proc. USENIX Annual Technical Conference*, pp. 79–90, 2001.

[29] "The zlib compression library." `http://www.zlib.net`.