Encrypted Integrity-Preserving Filesystem

Kaushal Patil(653639939)
Vasu Garg(676073104)

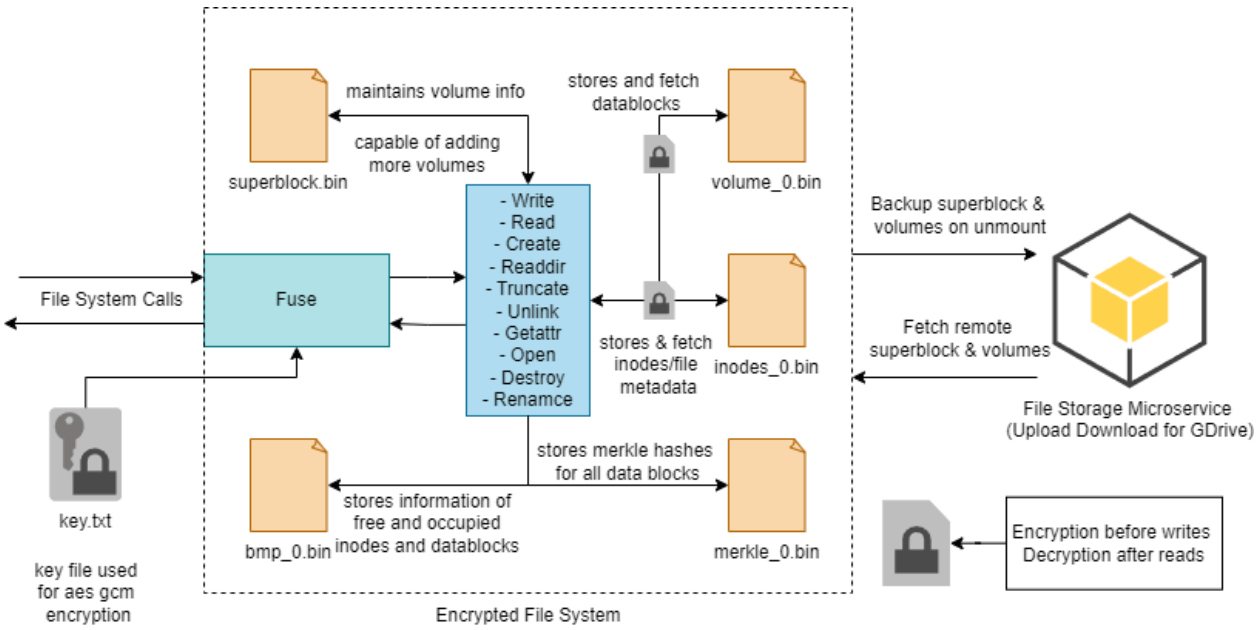
Introduction

EncryptFS explores the implementation of an encrypted filesystem using FUSE, integrated with the libsodium library, to utilize AES-GCM encryption for enhanced security, Merkle trees for integrity verification, and support for remote volume backup. It leverages cutting-edge cryptographic technologies and filesystem management techniques to offer security and flexibility. Designed with modern data security needs, EncryptFS provides robust tools to ensure data integrity, confidentiality, and availability, making it a valuable asset for businesses and individuals concerned with secure data management.

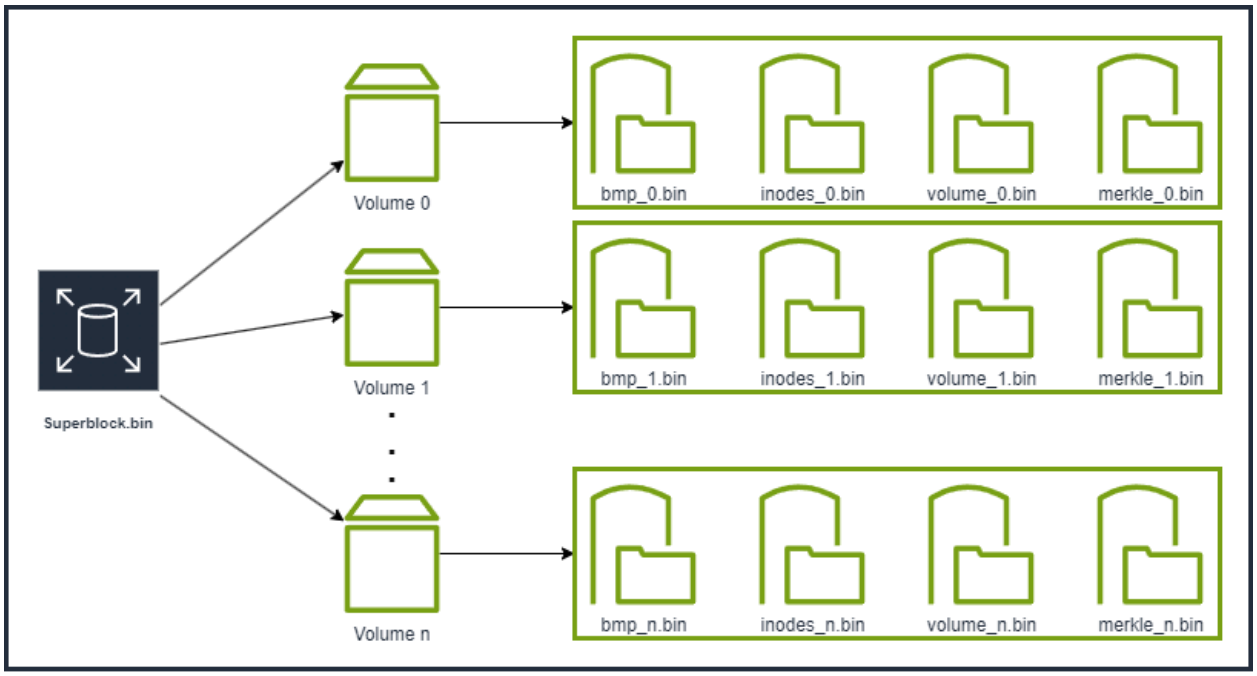
Key Features of EncryptFS

1. FUSE File System: Utilizes a Filesystem in Userspace (FUSE) to allow for user-mode file system development, enhancing security and customization without kernel modifications.
2. AES GCM Encryption: Implements AES encryption in Galois/Counter Mode (GCM) for data blocks and inodes, providing strong encryption and authentication.
    a. Optionally we also tried to check ChaChaPoly Encryption, the code base is compatible with it, provided there are a few minor changes.
3. Dynamically Expanding Volumes: Supports volume expansion dynamically, facilitating easy adaptation to increasing storage needs without service interruption.
4. Cloud Backup and Restoration: Enables secure backup of encrypted files to the cloud, supporting effective disaster recovery and data restoration strategies.
5. Merkle Tree for Integrity Verification: Merkle trees manage and verify data integrity, ensuring the data remains unaltered and secure against unauthorized changes.

System Design

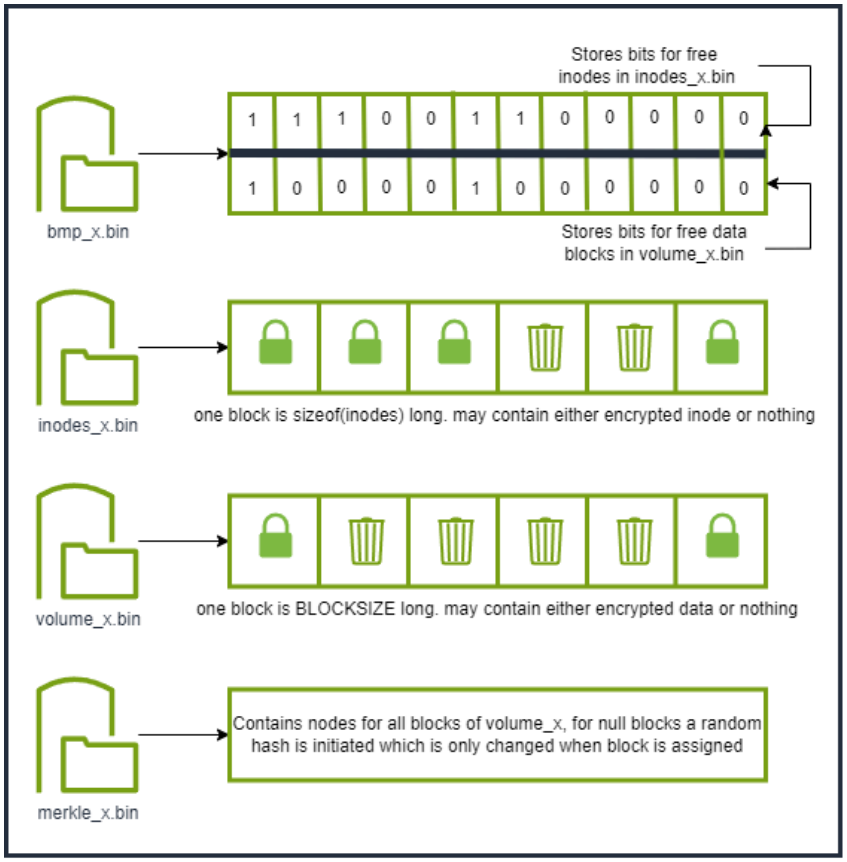


File System Structure



EncryptFS File System Data Structure

Storage Files Structure

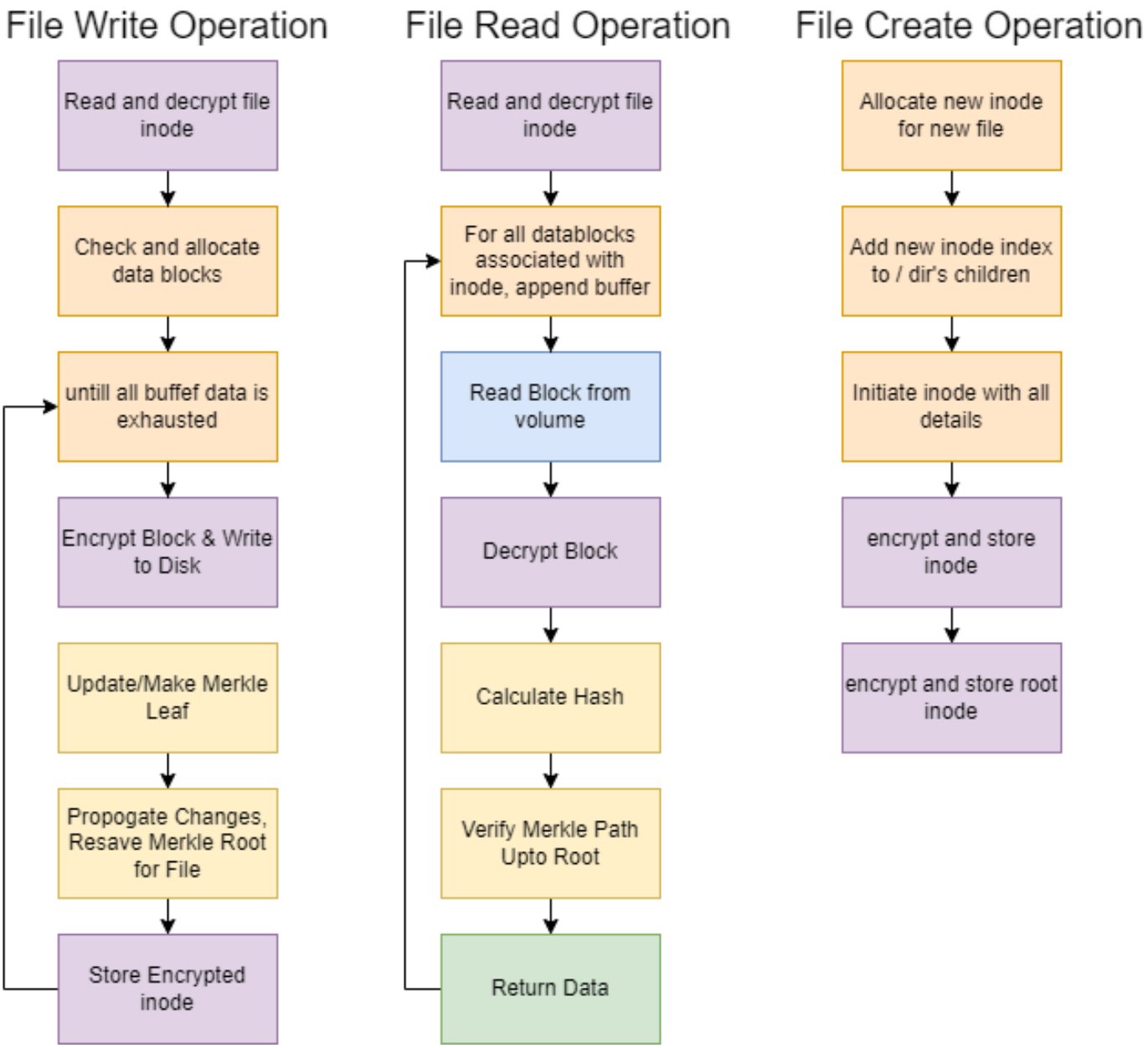


File Structure for encryptfs files

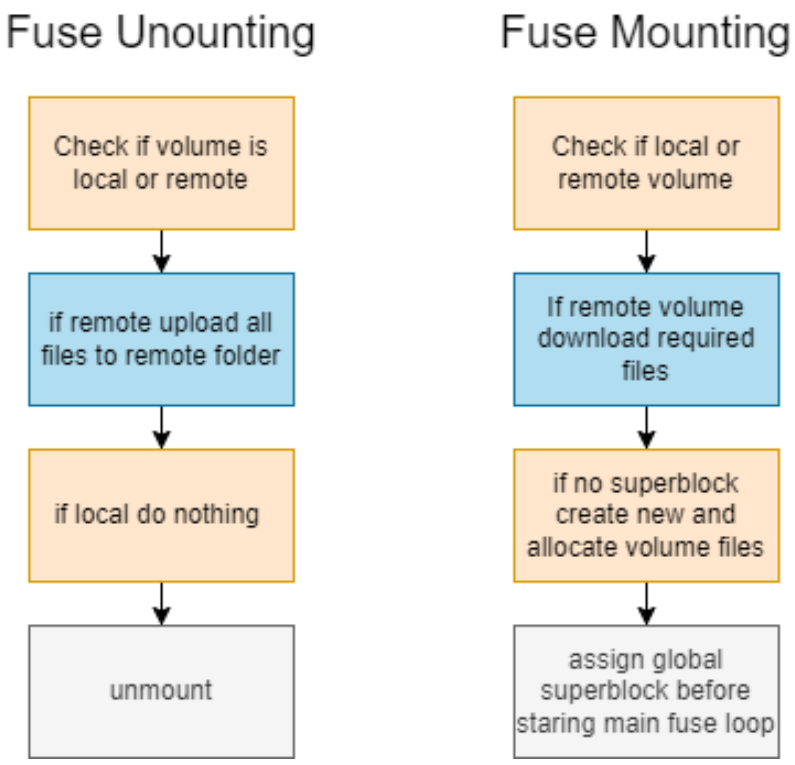
Algorithms:

EncryptFS encompasses a collection of algorithms designed to handle the creation, deletion, modification, and querying of files and directories(only a single root directory is currently supported) within the file system. This module ensures seamless integration and operation of file system commands, providing a robust interface for file management tasks. The algorithms are optimized for performance and reliability, ensuring efficient execution of file system operations while maintaining the integrity and security of the data stored in EncryptFS.

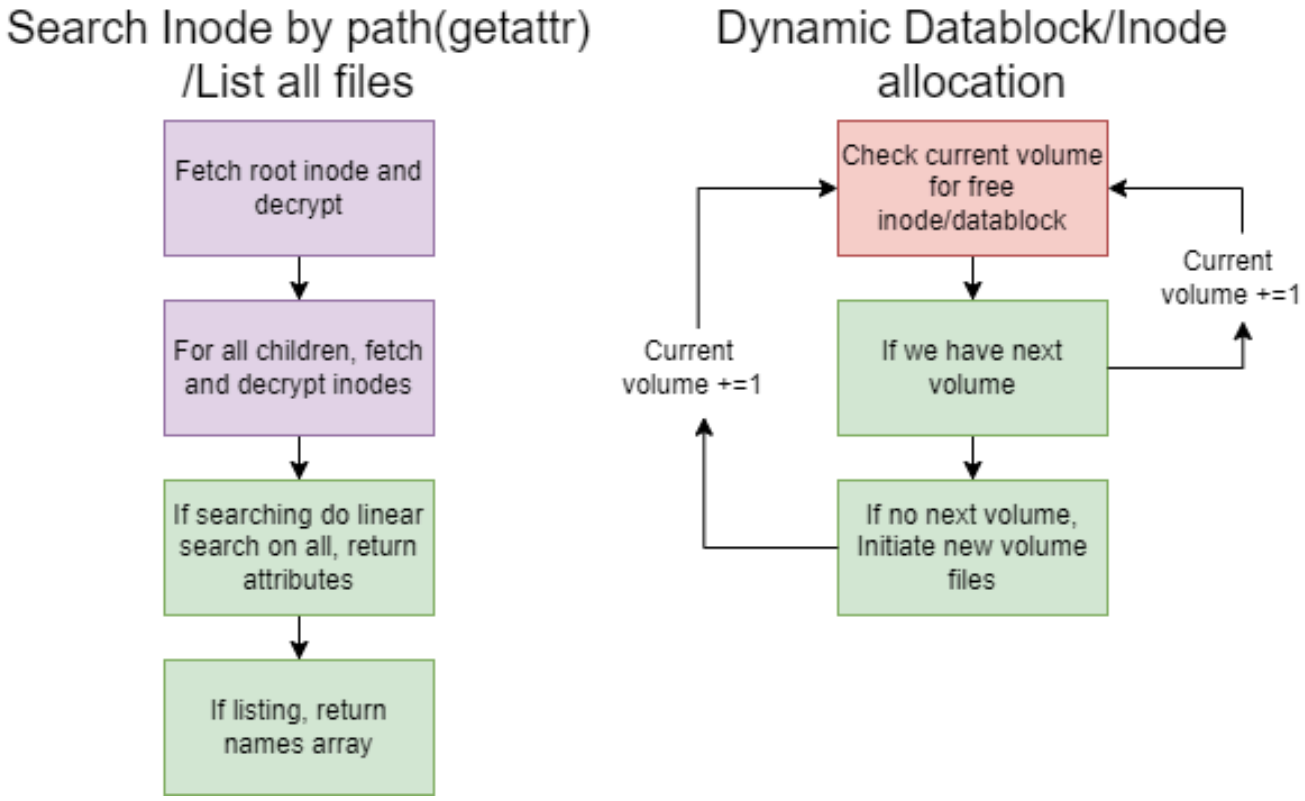
Create, Read, Write



Filesystem mounting and unmounting



File lookup/opening/readdir and Dynamic Volume Expansion



Architecture Components:

Superblock: The superblock contains the filesystem's information or metadata. When the filesystem is mounted, the superblock is the initialized structure. The following C structure defines the superblock used in the project. It contains volumes that can be dynamically expanded based on data needs

```
typedef struct superblock
{
   int volume_count;                  // Number of volumes
   int block_size;                    // Size of a block in bytes
   int inode_size;                    // Size of an inode in bytes
   volume_type vtype;                 // Type of volume
   volume_info_t volumes[NUMVOLUMES]; // Array of volume_info_t structures, Maximum defined in constants
} superblock_t;
```

Bitmap: In our FUSE project, a bitmap is used as an efficient method to manage the allocation of resources, such as data blocks or inodes, within the filesystem. Each bit in the bitmap represents the state of a resource unit (like a data block), indicating whether it is free or allocated.

Inodes: Each inode corresponds to a file or directory; in a sense the inode is that file or directory. Below is the inode struct where each inode includes details such as validity (valid), file path (path), permissions (permissions), file size (size), ownership (user_id, group_id), timestamps (a_time, m_time, c_time, b_time), and storage-related information like data block indices (datablocks), number of data blocks (num_datablocks), and hierarchical structure such as parent and child inodes (parent_inode, children, num_children), along with file type (type) and link count (num_links).

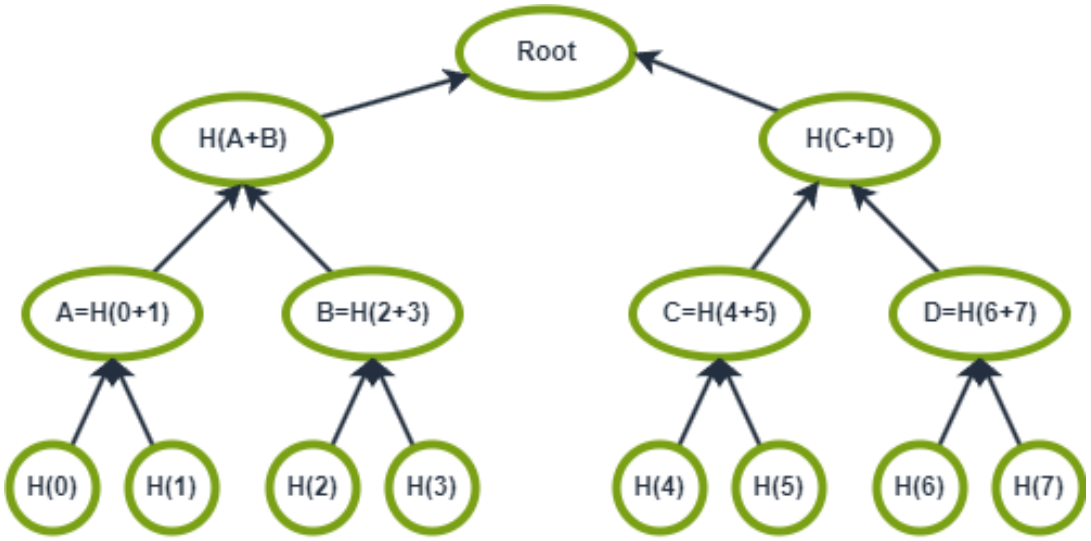
```
typedef struct inode
{
   int valid;                      // 0 if the inode is not valid, 1 if it is
   int inode_number;               // The inode number (Not used rn, but might be useful later)
   char path[MAX_PATH_LENGTH];     // The path of the file
   char name[MAX_NAME_LENGTH];     // The name of the file
   mode_t permissions;             // The permissions of the file
   bool is_directory;              // True if the inode is a directory, false if it is a file
   uid_t user_id;                  // The user id of the owner
   gid_t group_id;                 // The group id of the owner
   time_t a_time;                  // The last access time
   time_t m_time;                  // The last modification time
   time_t c_time;                  // The creation time
   time_t b_time;                  // The last time the inode was modified
   off_t size;                     // The size of the file
   int datablocks[MAX_DATABLOCKS]; // The data blocks that the file is stored in
   int num_datablocks;             // The number of data blocks that the file is stored in
   int parent_inode;               // The inode number of the parent directory
   int children[MAX_CHILDREN];     // Make sure MAX_CHILDREN is defined somewhere
   int num_children;               // The number of children in the directory
   char type[MAX_TYPE_LENGTH];     // The type of the file
   int num_links;                  // The number of links to the file
} inode;
```

FUSE functions: The following briefly describes all the API functions used in this project.

```
const struct fuse_operations fs_operations = {
   .getattr = fs_getattr,  // Get file attributes
   .open = fs_open,        // Open a file
   .readdir = fs_readdir,  // Read directory
   .rename = fs_rename,    // Rename a file
   .unlink = fs_unlink,    // Remove a file
   .create = fs_create,    // Create and Open a file
   .read = fs_read,        // Read data from an open file
   .write = fs_write,      // Write data to an open file
   .truncate = fs_truncate,// Change the size of a file
   .destroy = fs_destroy,  // The function that runs before file system is unmounted
};
```

Merkle Trees: In implementing the FUSE filesystem, we use Merkle trees to efficiently and securely verify the consistency and content of the data. A Merkle tree is a binary tree in which every leaf node is labeled with the cryptographic hash of a data block and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes.

Background on Merkle Trees: A Merkle tree, also known as a hash tree, is a data structure used primarily for verifying the content of large data structures, such as files or sets of files. The primary function of a Merkle tree is to efficiently and securely verify the consistency and content of the data. It is widely used in distributed systems like blockchains and file verification systems.



Merkle Tree Structure Built from data blocks

MerkleNode: Struct MerkleNode represents a single node in the tree containing a hash of the data, pointers to child nodes, and the range of block indices it covers.

```
typedef struct MerkleNode
{
   int block_index;           // Index of the block in the volume for searching purposes
   int min_index;             // Minimum index in the subtree
   int max_index;             // Maximum index in the subtree
   char hash[65];             // Hash stored in this node
   struct MerkleNode *left;   // Pointer to left child
   struct MerkleNode *right;  // Pointer to right child
   struct MerkleNode *parent; // Pointer to parent node
} MerkleNode;
```

The following functions outline the implementation of a Merkle tree for the FUSE filesystem:

1. Merkle Tree Construction and Management:

- `build_merkle_tree()`: Builds a Merkle tree from an array of block hashes. This function initializes the tree by creating nodes from the bottom up until the root is reached.

- `initialize_merkle_tree_for_volume()`: Initializes a Merkle tree for a specific volume, likely by hashing all the blocks within that volume and constructing the tree.

- `save_merkle_tree_to_file()`: Saves the Merkle tree to a file, which is useful for persisting the tree state between system reboots or for backup purposes.

- `load_merkle_tree_from_file()`: Loads a Merkle tree from a file, restoring its state to maintain continuity in integrity verification across sessions.

2. Verification and Integrity Checks:

- `verify_merkle_path()`: Verifies that the hash path from a leaf node up to the root matches the expected root hash, confirming the integrity of a block or a set of blocks within the tree.

- `verify_block_integrity()`: Integrates with the Merkle tree to confirm the integrity of a specific block, using its index to find the corresponding leaf in the tree and then verifying the path to the root.

3. Persistence:

- `save_merkle_tree_to_file()`: Serializes the Merkle tree to a file to preserve its state between sessions.

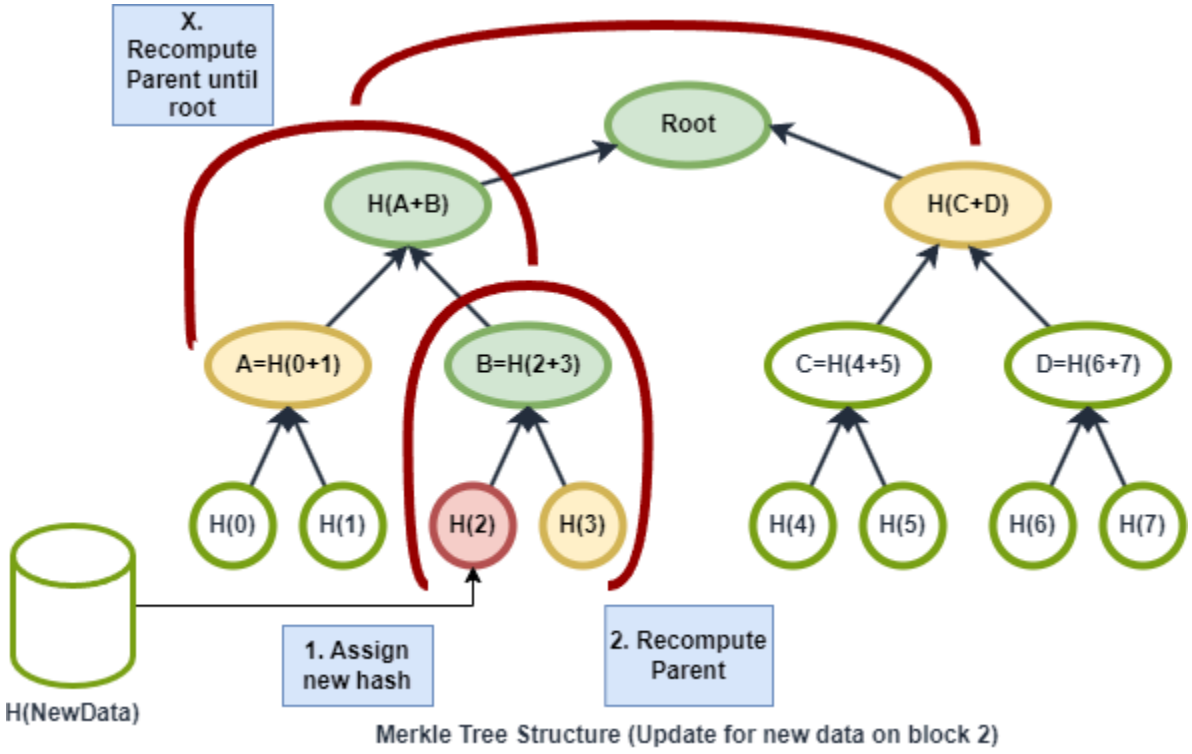
- `load_merkle_tree_from_file()`: Deserializes the Merkle tree from a file, enabling persistent integrity checking across system reboots or sessions.

4. Utility Operations:

- `get_block_hash()`, `generate_random_hash()`: Manage and generate hashes for blocks, critical for tree construction and updates.

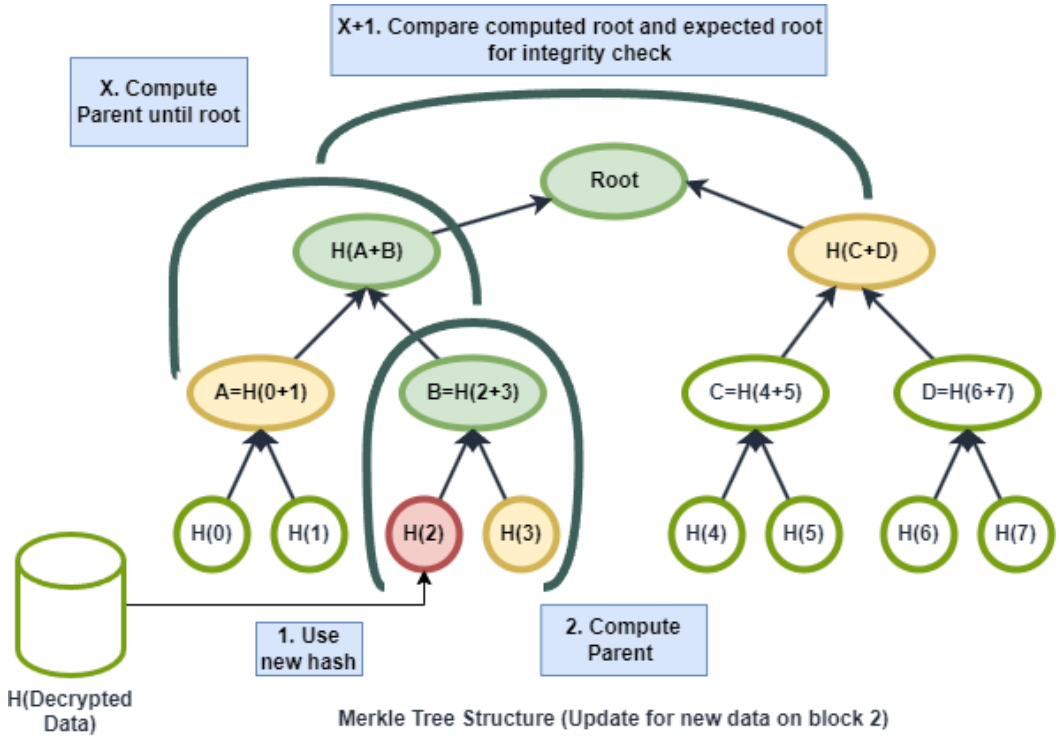
- `get_root_hash()`: Retrieves the current root hash of the tree, crucial for verifying the overall integrity of the dataset.

Merkle Tree Update:



Merkle Tree Structure (Update for new data on block 2)

Merkle Tree Verification:



Merkle Tree Structure (Update for new data on block 2)

AES/GCM Encryption:

In our project, AES-GCM is implemented using the Libsodium library, which offers a high-level toolset for cryptographic operations including AES-GCM. Here's a summary of how AES-GCM is being used:

Encryption Process

1. Key and Nonce Generation: A 256-bit symmetric key is generated or loaded from an external source, depending on the executed function. This key is crucial for both encryption and decryption processes.

A nonce (number used once) is generated for each encryption operation to ensure uniqueness. In AES-GCM, the nonce is critical because reusing a nonce with the same key can compromise the security of the encryption.

2. Encrypting Data: When data (such as an inode or a volume block) needs to be encrypted before writing to disk, it is passed along with the generated nonce and the key to the `encrypt_aes_gcm()` function.

Inside the `encrypt_aes_gcm(),` the `crypto_aead_aes256gcm_encrypt()` function from Libsodium performs the encryption. It outputs the ciphertext (encrypted data) and its length. This function also ensures that the data is authenticated (checked for integrity and authenticity), which means changes to the data can be detected.

3. Storing Encrypted Data: The resulting ciphertext and the nonce (required for decryption) are written to disk. This typically happens in the file associated with the data being encrypted (e.g., inode file or volume file).

Decryption Process

1. Loading Nonce and Ciphertext: When reading the encrypted data from disk, the stored nonce and ciphertext are first loaded from the respective file.

2. Decrypting Data: The `decrypt_aes_gcm()` function is used to decrypt the data. It takes the loaded ciphertext and nonce, along with the symmetric key.

The function `crypto_aead_aes256gcm_decrypt()` from Libsodium is called internally, which attempts to decrypt the ciphertext back into plaintext and also verify its integrity using the authentication tag included in the ciphertext.

3. Handling Decryption Output: If decryption is successful and the data is verified as authentic and unaltered, the plaintext is written into the buffer provided to the decryption function.

If decryption fails (for reasons such as incorrect nonce, key, or data tampering), an error is reported, and appropriate actions can be taken (such as error logging or throwing exceptions).