fs-verity: read-only file-based authenticity protection

Introduction

fs-verity (`fs/verity/`) is a support layer that filesystems can hook into to support transparent integrity and authenticity protection of read-only files. Currently, it is supported by the ext4 and f2fs filesystems. Like fscrypt, not too much filesystem-specific code is needed to support fs-verity.

fs-verity is similar to dm-verity but works on files rather than block devices. On regular files on filesystems supporting fs-verity, userspace can execute an ioctl that causes the filesystem to build a Merkle tree for the file and persist it to a filesystem-specific location associated with the file.

After this, the file is made readonly, and all reads from the file are automatically verified against the file's Merkle tree. Reads of any corrupted data, including mmap reads, will fail.

Userspace can use another ioctl to retrieve the root hash (actually the "fs-verity file digest", which is a hash that includes the Merkle tree root hash) that fs-verity is enforcing for the file. This ioctl executes in constant time, regardless of the file size.

fs-verity is essentially a way to hash a file in constant time, subject to the caveat that reads which would violate the hash will fail at runtime.

Use cases

By itself, the base fs-verity feature only provides integrity protection, i.e. detection of accidental (non-malicious) corruption.

However, because fs-verity makes retrieving the file hash extremely efficient, it's primarily meant to be used as a tool to support authentication (detection of malicious modifications) or auditing (logging file hashes before use).

Trusted userspace code (e.g. operating system code running on a read-only partition that is itself authenticated by dm-verity) can authenticate the contents of an fs-verity file by using the FS_IOC_MEASURE_VERITY ioctl to retrieve its hash, then verifying a digital signature of it.

A standard file hash could be used instead of fs-verity. However, this is inefficient if the file is large and only a small portion may be accessed. This is often the case for Android application package (APK) files, for example. These typically contain many translations, classes, and other resources that are infrequently or even never accessed on a particular device. It would be slow and wasteful to read and hash the entire file before starting the application.

Unlike an ahead-of-time hash, fs-verity also re-verifies data each time it's paged in. This ensures that malicious disk firmware can't undetectably change the contents of the file at runtime.

fs-verity does not replace or obsolete dm-verity. dm-verity should still be used on read-only filesystems. fs-verity is for files that must live on a read-write filesystem because they are independently updated and potentially user-installed, so dm-verity cannot be used.

The base fs-verity feature is a hashing mechanism only; actually authenticating the files is up to userspace. However, to meet some users' needs, fs-verity optionally supports a simple signature verification mechanism where users can configure the kernel to require that all fs-verity files be signed by a key loaded into a keyring; see Built-in signature verification. Support for fs-verity file hashes in IMA (Integrity Measurement Architecture) policies is also planned.

User API

FS_IOC_ENABLE_VERITY

The FS_IOC_ENABLE_VERITY ioctl enables fs-verity on a file. It takes in a pointer to a struct fsverity_enable_arg, defined as follows:

```
struct fsverity_enable_arg {
        __u32 version;
        __u32 hash_algorithm;
        __u32 block_size;
        __u32 salt_size;
        __u64 salt_ptr;
        __u32 sig_size;
        __u32 __reserved1;
        __u64 sig_ptr;
        __u64 __reserved2[11];
};
```

This structure contains the parameters of the Merkle tree to build for the file, and optionally contains a signature. It must be initialized as follows:

- `version` must be 1.
- `hash_algorithm` must be the identifier for the hash algorithm to use for the Merkle tree, such as

FS_VERITY_HASH_ALG_SHA256. See `include/uapi/linux/fsverity.h` for the list of possible values.
- `block_size` must be the Merkle tree block size. Currently, this must be equal to the system page size, which is usually 4096 bytes. Other sizes may be supported in the future. This value is not necessarily the same as the filesystem block size.
- `salt_size` is the size of the salt in bytes, or 0 if no salt is provided. The salt is a value that is prepended to every hashed block; it can be used to personalize the hashing for a particular file or device. Currently the maximum salt size is 32 bytes.
- `salt_ptr` is the pointer to the salt, or NULL if no salt is provided.
- `sig_size` is the size of the signature in bytes, or 0 if no signature is provided. Currently the signature is (somewhat arbitrarily) limited to 16128 bytes. See Built-in signature verification for more information.
- `sig_ptr` is the pointer to the signature, or NULL if no signature is provided.
- All reserved fields must be zeroed.

FS_IOC_ENABLE_VERITY causes the filesystem to build a Merkle tree for the file and persist it to a filesystem-specific location associated with the file, then mark the file as a verity file. This ioctl may take a long time to execute on large files, and it is interruptible by fatal signals.

FS_IOC_ENABLE_VERITY checks for write access to the inode. However, it must be executed on an O_RDONLY file descriptor and no processes can have the file open for writing. Attempts to open the file for writing while this ioctl is executing will fail with ETXTBSY. (This is necessary to guarantee that no writable file descriptors will exist after verity is enabled, and to guarantee that the file's contents are stable while the Merkle tree is being built over it.)

On success, FS_IOC_ENABLE_VERITY returns 0, and the file becomes a verity file. On failure (including the case of interruption by a fatal signal), no changes are made to the file.

FS_IOC_ENABLE_VERITY can fail with the following errors:

- `EACCES`: the process does not have write access to the file
- `EBADMSG`: the signature is malformed
- `EBUSY`: this ioctl is already running on the file
- `EEXIST`: the file already has verity enabled
- `EFAULT`: the caller provided inaccessible memory
- `EINTR`: the operation was interrupted by a fatal signal
- `EINVAL`: unsupported version, hash algorithm, or block size; or reserved bits are set; or the file descriptor refers to neither a regular file nor a directory.
- `EISDIR`: the file descriptor refers to a directory
- `EKEYREJECTED`: the signature doesn't match the file
- `EMSGSIZE`: the salt or signature is too long
- `ENOKEY`: the fs-verity keyring doesn't contain the certificate needed to verify the signature
- `ENOPKG`: fs-verity recognizes the hash algorithm, but it's not available in the kernel's crypto API as currently configured (e.g. for SHA-512, missing CONFIG_CRYPTO_SHA512).
- `ENOTTY`: this type of filesystem does not implement fs-verity
- `EOPNOTSUPP`: the kernel was not configured with fs-verity support; or the filesystem superblock has not had the 'verity' feature enabled on it; or the filesystem does not support fs-verity on this file. (See Filesystem support.)
- `EPERM`: the file is append-only; or, a signature is required and one was not provided.
- `EROFS`: the filesystem is read-only
- `ETXTBSY`: someone has the file open for writing. This can be the caller's file descriptor, another open file descriptor, or the file reference held by a writable memory map.

FS_IOC_MEASURE_VERITY

The FS_IOC_MEASURE_VERITY ioctl retrieves the digest of a verity file. The fs-verity file digest is a cryptographic digest that identifies the file contents that are being enforced on reads; it is computed via a Merkle tree and is different from a traditional full-file digest.

This ioctl takes in a pointer to a variable-length structure:

```
struct fsverity_digest {
        __u16 digest_algorithm;
        __u16 digest_size; /* input/output */
        __u8 digest[];
};
```

`digest_size` is an input/output field. On input, it must be initialized to the number of bytes allocated for the variable-length `digest` field.

On success, 0 is returned and the kernel fills in the structure as follows:

- `digest_algorithm` will be the hash algorithm used for the file digest. It will match `fsverity_enable_arg::hash_algorithm`.
- `digest_size` will be the size of the digest in bytes, e.g. 32 for SHA-256. (This can be redundant with `digest_algorithm`.)
- `digest` will be the actual bytes of the digest.

FS_IOC_MEASURE_VERITY is guaranteed to execute in constant time, regardless of the size of the file.

FS_IOC_MEASURE_VERITY can fail with the following errors:

- `EFAULT`: the caller provided inaccessible memory
- `ENODATA`: the file is not a verity file
- `ENOTTY`: this type of filesystem does not implement fs-verity
- `EOPNOTSUPP`: the kernel was not configured with fs-verity support, or the filesystem superblock has not had the 'verity' feature enabled on it. (See Filesystem support.)
- `EOVERFLOW`: the digest is longer than the specified `digest_size` bytes. Try providing a larger buffer.

FS_IOC_READ_VERITY_METADATA

The FS_IOC_READ_VERITY_METADATA ioctl reads verity metadata from a verity file. This ioctl is available since Linux v5.12.

This ioctl allows writing a server program that takes a verity file and serves it to a client program, such that the client can do its own fs-verity compatible verification of the file. This only makes sense if the client doesn't trust the server and if the server needs to provide the storage for the client.

This is a fairly specialized use case, and most fs-verity users won't need this ioctl.

This ioctl takes in a pointer to the following structure:

```
#define FS_VERITY_METADATA_TYPE_MERKLE_TREE     1
#define FS_VERITY_METADATA_TYPE_DESCRIPTOR      2
#define FS_VERITY_METADATA_TYPE_SIGNATURE       3

struct fsverity_read_metadata_arg {
        __u64 metadata_type;
        __u64 offset;
        __u64 length;
        __u64 buf_ptr;
        __u64 __reserved;
};
```

`metadata_type` specifies the type of metadata to read:

- `FS_VERITY_METADATA_TYPE_MERKLE_TREE` reads the blocks of the Merkle tree. The blocks are returned in order from the root level to the leaf level. Within each level, the blocks are returned in the same order that their hashes are themselves hashed. See Merkle tree for more information.
- `FS_VERITY_METADATA_TYPE_DESCRIPTOR` reads the fs-verity descriptor. See fs-verity descriptor.
- `FS_VERITY_METADATA_TYPE_SIGNATURE` reads the signature which was passed to FS_IOC_ENABLE_VERITY, if any. See Built-in signature verification.

The semantics are similar to those of `pread()`. `offset` specifies the offset in bytes into the metadata item to read from, and `length` specifies the maximum number of bytes to read from the metadata item. `buf_ptr` is the pointer to the buffer to read into, cast to a 64-bit integer. `__reserved` must be 0. On success, the number of bytes read is returned. 0 is returned at the end of the metadata item. The returned length may be less than `length`, for example if the ioctl is interrupted.

The metadata returned by FS_IOC_READ_VERITY_METADATA isn't guaranteed to be authenticated against the file digest that would be returned by FS_IOC_MEASURE_VERITY, as the metadata is expected to be used to implement fs-verity compatible verification anyway (though absent a malicious disk, the metadata will indeed match). E.g. to implement this ioctl, the filesystem is allowed to just read the Merkle tree blocks from disk without actually verifying the path to the root node.

FS_IOC_READ_VERITY_METADATA can fail with the following errors:

- `EFAULT`: the caller provided inaccessible memory
- `EINTR`: the ioctl was interrupted before any data was read
- `EINVAL`: reserved fields were set, or `offset + length` overflowed
- `ENODATA`: the file is not a verity file, or FS_VERITY_METADATA_TYPE_SIGNATURE was requested but the file doesn't have a built-in signature
- `ENOTTY`: this type of filesystem does not implement fs-verity, or this ioctl is not yet implemented on it
- `EOPNOTSUPP`: the kernel was not configured with fs-verity support, or the filesystem superblock has not had the 'verity' feature enabled on it. (See Filesystem support.)

FS_IOC_GETFLAGS

The existing ioctl FS_IOC_GETFLAGS (which isn't specific to fs-verity) can also be used to check whether a file has fs-verity enabled or not. To do so, check for FS_VERITY_FL (0x00100000) in the returned flags.

The verity flag is not settable via FS_IOC_SETFLAGS. You must use FS_IOC_ENABLE_VERITY instead, since parameters must be provided.

statx

Since Linux v5.5, the statx() system call sets STATX_ATTR_VERITY if the file has fs-verity enabled. This can perform better than FS_IOC_GETFLAGS and FS_IOC_MEASURE_VERITY because it doesn't require opening the file, and opening verity files can

be expensive.

Accessing verity files

Applications can transparently access a verity file just like a non-verity one, with the following exceptions:

- Verity files are readonly. They cannot be opened for writing or truncate()d, even if the file mode bits allow it. Attempts to do one of these things will fail with EPERM. However, changes to metadata such as owner, mode, timestamps, and xattrs are still allowed, since these are not measured by fs-verity. Verity files can also still be renamed, deleted, and linked to.
- Direct I/O is not supported on verity files. Attempts to use direct I/O on such files will fall back to buffered I/O.
- DAX (Direct Access) is not supported on verity files, because this would circumvent the data verification.
- Reads of data that doesn't match the verity Merkle tree will fail with EIO (for read()) or SIGBUS (for mmap() reads).
- If the sysctl "fs.verity.require_signatures" is set to 1 and the file is not signed by a key in the fs-verity keyring, then opening the file will fail. See Built-in signature verification.

Direct access to the Merkle tree is not supported. Therefore, if a verity file is copied, or is backed up and restored, then it will lose its "verity"-ness. fs-verity is primarily meant for files like executables that are managed by a package manager.

File digest computation

This section describes how fs-verity hashes the file contents using a Merkle tree to produce the digest which cryptographically identifies the file contents. This algorithm is the same for all filesystems that support fs-verity.

Userspace only needs to be aware of this algorithm if it needs to compute fs-verity file digests itself, e.g. in order to sign files.

Merkle tree

The file contents is divided into blocks, where the block size is configurable but is usually 4096 bytes. The end of the last block is zero-padded if needed. Each block is then hashed, producing the first level of hashes. Then, the hashes in this first level are grouped into 'blocksize'-byte blocks (zero-padding the ends as needed) and these blocks are hashed, producing the second level of hashes. This proceeds up the tree until only a single block remains. The hash of this block is the "Merkle tree root hash".

If the file fits in one block and is nonempty, then the "Merkle tree root hash" is simply the hash of the single data block. If the file is empty, then the "Merkle tree root hash" is all zeroes.

The "blocks" here are not necessarily the same as "filesystem blocks".

If a salt was specified, then it's zero-padded to the closest multiple of the input size of the hash algorithm's compression function, e.g. 64 bytes for SHA-256 or 128 bytes for SHA-512. The padded salt is prepended to every data or Merkle tree block that is hashed.

The purpose of the block padding is to cause every hash to be taken over the same amount of data, which simplifies the implementation and keeps open more possibilities for hardware acceleration. The purpose of the salt padding is to make the salting "free" when the salted hash state is precomputed, then imported for each hash.

Example: in the recommended configuration of SHA-256 and 4K blocks, 128 hash values fit in each block. Thus, each level of the Merkle tree is approximately 128 times smaller than the previous, and for large files the Merkle tree's size converges to approximately 1/127 of the original file size. However, for small files, the padding is significant, making the space overhead proportionally more.

fs-verity descriptor

By itself, the Merkle tree root hash is ambiguous. For example, it can't a distinguish a large file from a small second file whose data is exactly the top-level hash block of the first file. Ambiguities also arise from the convention of padding to the next block boundary.

To solve this problem, the fs-verity file digest is actually computed as a hash of the following structure, which contains the Merkle tree root hash as well as other fields such as the file size:

```
struct fsverity_descriptor {
        __u8 version;           /* must be 1 */
        __u8 hash_algorithm;    /* Merkle tree hash algorithm */
        __u8 log_blocksize;     /* log2 of size of data and tree blocks */
        __u8 salt_size;         /* size of salt in bytes; 0 if none */
        __le32 __reserved_0x04; /* must be 0 */
        __le64 data_size;       /* size of file the Merkle tree is built over */
        __u8 root_hash[64];     /* Merkle tree root hash */
        __u8 salt[32];          /* salt prepended to each hashed block */
        __u8 __reserved[144];   /* must be 0's */
};
```

Built-in signature verification

With CONFIG_FS_VERITY_BUILTIN_SIGNATURES=y, fs-verity supports putting a portion of an authentication policy (see Use cases) in the kernel. Specifically, it adds support for:

1. At fs-verity module initialization time, a keyring ".fs-verity" is created. The root user can add trusted X.509 certificates to this keyring using the add_key() system call, then (when done) optionally use keyctl_restrict_keyring() to prevent additional certificates from being added.
2. FS_IOC_ENABLE_VERITY accepts a pointer to a PKCS#7 formatted detached signature in DER format of the file's fs-verity digest. On success, this signature is persisted alongside the Merkle tree. Then, any time the file is opened, the kernel will verify the file's actual digest against this signature, using the certificates in the ".fs-verity" keyring.
3. A new sysctl "fs.verity.require_signatures" is made available. When set to 1, the kernel requires that all verity files have a correctly signed digest as described in (2).

fs-verity file digests must be signed in the following format, which is similar to the structure used by FS_IOC_MEASURE_VERITY:

```
struct fsverity_formatted_digest {
        char magic[8];                  /* must be "FSVerity" */
        __le16 digest_algorithm;
        __le16 digest_size;
        __u8 digest[];
};
```

fs-verity's built-in signature verification support is meant as a relatively simple mechanism that can be used to provide some level of authenticity protection for verity files, as an alternative to doing the signature verification in userspace or using IMA-appraisal. However, with this mechanism, userspace programs still need to check that the verity bit is set, and there is no protection against verity files being swapped around.

Filesystem support

fs-verity is currently supported by the ext4 and f2fs filesystems. The CONFIG_FS_VERITY kconfig option must be enabled to use fs-verity on either filesystem.

`include/linux/fsverity.h` declares the interface between the `fs/verity/` support layer and filesystems. Briefly, filesystems must provide an `fsverity_operations` structure that provides methods to read and write the verity metadata to a filesystem-specific location, including the Merkle tree blocks and `fsverity_descriptor`. Filesystems must also call functions in `fs/verity/` at certain times, such as when a file is opened or when pages have been read into the pagecache. (See Verifying data.)

ext4

ext4 supports fs-verity since Linux v5.4 and e2fsprogs v1.45.2.

To create verity files on an ext4 filesystem, the filesystem must have been formatted with `-O verity` or had `tune2fs -O verity` run on it. "verity" is an RO_COMPAT filesystem feature, so once set, old kernels will only be able to mount the filesystem readonly, and old versions of e2fsck will be unable to check the filesystem. Moreover, currently ext4 only supports mounting a filesystem with the "verity" feature when its block size is equal to PAGE_SIZE (often 4096 bytes).

ext4 sets the EXT4_VERITY_FL on-disk inode flag on verity files. It can only be set by FS_IOC_ENABLE_VERITY, and it cannot be cleared.

ext4 also supports encryption, which can be used simultaneously with fs-verity. In this case, the plaintext data is verified rather than the ciphertext. This is necessary in order to make the fs-verity file digest meaningful, since every file is encrypted differently.

ext4 stores the verity metadata (Merkle tree and fsverity_descriptor) past the end of the file, starting at the first 64K boundary beyond i_size. This approach works because (a) verity files are readonly, and (b) pages fully beyond i_size aren't visible to userspace but can be read/written internally by ext4 with only some relatively small changes to ext4. This approach avoids having to depend on the EA_INODE feature and on rearchitecturing ext4's xattr support to support paging multi-gigabyte xattrs into memory, and to support encrypting xattrs. Note that the verity metadata *must* be encrypted when the file is, since it contains hashes of the plaintext data.

Currently, ext4 verity only supports the case where the Merkle tree block size, filesystem block size, and page size are all the same. It also only supports extent-based files.

f2fs

f2fs supports fs-verity since Linux v5.4 and f2fs-tools v1.11.0.

To create verity files on an f2fs filesystem, the filesystem must have been formatted with `-O verity`.

f2fs sets the FADVISE_VERITY_BIT on-disk inode flag on verity files. It can only be set by FS_IOC_ENABLE_VERITY, and it cannot be cleared.

Like ext4, f2fs stores the verity metadata (Merkle tree and fsverity_descriptor) past the end of the file, starting at the first 64K boundary beyond i_size. See explanation for ext4 above. Moreover, f2fs supports at most 4096 bytes of xattr entries per inode which wouldn't be enough for even a single Merkle tree block.

Currently, f2fs verity only supports a Merkle tree block size of 4096. Also, f2fs doesn't support enabling verity on files that currently have atomic or volatile writes pending.

Implementation details

Implementation details

Verifying data

fs-verity ensures that all reads of a verity file's data are verified, regardless of which syscall is used to do the read (e.g. mmap(), read(), pread()) and regardless of whether it's the first read or a later read (unless the later read can return cached data that was already verified). Below, we describe how filesystems implement this.

Pagecache

For filesystems using Linux's pagecache, the `->readpage()` and `->readahead()` methods must be modified to verify pages before they are marked Uptodate. Merely hooking `->read_iter()` would be insufficient, since `->read_iter()` is not used for memory maps.

Therefore, fs/verity/ provides a function fsverity_verify_page() which verifies a page that has been read into the pagecache of a verity inode, but is still locked and not Uptodate, so it's not yet readable by userspace. As needed to do the verification, fsverity_verify_page() will call back into the filesystem to read Merkle tree pages via fsverity_operations::read_merkle_tree_page().

fsverity_verify_page() returns false if verification failed; in this case, the filesystem must not set the page Uptodate. Following this, as per the usual Linux pagecache behavior, attempts by userspace to read() from the part of the file containing the page will fail with EIO, and accesses to the page within a memory map will raise SIGBUS.

fsverity_verify_page() currently only supports the case where the Merkle tree block size is equal to PAGE_SIZE (often 4096 bytes).

In principle, fsverity_verify_page() verifies the entire path in the Merkle tree from the data page to the root hash. However, for efficiency the filesystem may cache the hash pages. Therefore, fsverity_verify_page() only ascends the tree reading hash pages until an already-verified hash page is seen, as indicated by the PageChecked bit being set. It then verifies the path to that page.

This optimization, which is also used by dm-verity, results in excellent sequential read performance. This is because usually (e.g. 127 in 128 times for 4K blocks and SHA-256) the hash page from the bottom level of the tree will already be cached and checked from reading a previous data page. However, random reads perform worse.

Block device based filesystems

Block device based filesystems (e.g. ext4 and f2fs) in Linux also use the pagecache, so the above subsection applies too. However, they also usually read many pages from a file at once, grouped into a structure called a "bio". To make it easier for these types of filesystems to support fs-verity, fs/verity/ also provides a function fsverity_verify_bio() which verifies all pages in a bio.

ext4 and f2fs also support encryption. If a verity file is also encrypted, the pages must be decrypted before being verified. To support this, these filesystems allocate a "post-read context" for each bio and store it in `->bi_private`:

```
struct bio_post_read_ctx {
        struct bio *bio;
        struct work_struct work;
        unsigned int cur_step;
        unsigned int enabled_steps;
};
```

`enabled_steps` is a bitmask that specifies whether decryption, verity, or both is enabled. After the bio completes, for each needed postprocessing step the filesystem enqueues the bio_post_read_ctx on a workqueue, and then the workqueue work does the decryption or verification. Finally, pages where no decryption or verity error occurred are marked Uptodate, and the pages are unlocked.

Files on ext4 and f2fs may contain holes. Normally, `->readahead()` simply zeroes holes and sets the corresponding pages Uptodate; no bios are issued. To prevent this case from bypassing fs-verity, these filesystems use fsverity_verify_page() to verify hole pages.

ext4 and f2fs disable direct I/O on verity files, since otherwise direct I/O would bypass fs-verity. (They also do the same for encrypted files.)

Userspace utility

This document focuses on the kernel, but a userspace utility for fs-verity can be found at:

https://git.kernel.org/pub/scm/linux/kernel/git/ebiggers/fsverity-utils.git

See the README.md file in the fsverity-utils source tree for details, including examples of setting up fs-verity protected files.

Tests

To test fs-verity, use xfstests. For example, using kvm-xfstests:

```
kvm-xfstests -c ext4,f2fs -g verity
```

FAQ

This section answers frequently asked questions about fs-verity that weren't already directly answered in other parts of this document.

Q: Why isn't fs-verity part of IMA?

A: fs-verity and IMA (Integrity Measurement Architecture) have different focuses. fs-verity is a filesystem-level mechanism for hashing individual files using a Merkle tree. In contrast, IMA specifies a system-wide policy that specifies which files are hashed and what to do with those hashes, such as log them, authenticate them, or add them to a measurement list.

IMA is planned to support the fs-verity hashing mechanism as an alternative to doing full file hashes, for people who want the performance and security benefits of the Merkle tree based hash. But it doesn't make sense to force all uses of fs-verity to be through IMA. As a standalone filesystem feature, fs-verity already meets many users' needs, and it's testable like other filesystem features e.g. with xfstests.

Q: Isn't fs-verity useless because the attacker can just modify the hashes in the Merkle tree, which is stored on-disk?

A: To verify the authenticity of an fs-verity file you must verify the authenticity of the "fs-verity file digest", which incorporates the root hash of the Merkle tree. See Use cases.

Q: Isn't fs-verity useless because the attacker can just replace a verity file with a non-verity one?

A: See Use cases. In the initial use case, it's really trusted userspace code that authenticates the files; fs-verity is just a tool to do this job efficiently and securely. The trusted userspace code will consider non-verity files to be inauthentic.

Q: Why does the Merkle tree need to be stored on-disk? Couldn't you store just the root hash?

A: If the Merkle tree wasn't stored on-disk, then you'd have to compute the entire tree when the file is first accessed, even if just one byte is being read. This is a fundamental consequence of how Merkle tree hashing works. To verify a leaf node, you need to verify the whole path to the root hash, including the root node (the thing which the root hash is a hash of). But if the root node isn't stored on-disk, you have to compute it by hashing its children, and so on until you've actually hashed the entire file.

That defeats most of the point of doing a Merkle tree-based hash, since if you have to hash the whole file ahead of time anyway, then you could simply do sha256(file) instead. That would be much simpler, and a bit faster too.

It's true that an in-memory Merkle tree could still provide the advantage of verification on every read rather than just on the first read. However, it would be inefficient because every time a hash page gets evicted (you can't pin the entire Merkle tree into memory, since it may be very large), in order to restore it you again need to hash everything below it in the tree. This again defeats most of the point of doing a Merkle tree-based hash, since a single block read could trigger re-hashing gigabytes of data.

Q: But couldn't you store just the leaf nodes and compute the rest?

A: See previous answer; this really just moves up one level, since one could alternatively interpret the data blocks as being the leaf nodes of the Merkle tree. It's true that the tree can be computed much faster if the leaf level is stored rather than just the data, but that's only because each level is less than 1% the size of the level below (assuming the recommended settings of SHA-256 and 4K blocks). For the exact same reason, by storing "just the leaf nodes" you'd already be storing over 99% of the tree, so you might as well simply store the whole tree.

Q: Can the Merkle tree be built ahead of time, e.g. distributed as part of a package that is installed to many computers?

A: This isn't currently supported. It was part of the original design, but was removed to simplify the kernel UAPI and because it wasn't a critical use case. Files are usually installed once and used many times, and cryptographic hashing is somewhat fast on most modern processors.

Q: Why doesn't fs-verity support writes?

A: Write support would be very difficult and would require a completely different design, so it's well outside the scope of fs-verity. Write support would require:

- A way to maintain consistency between the data and hashes, including all levels of hashes, since corruption after a crash (especially of potentially the entire file!) is unacceptable. The main options for solving this are data journalling, copy-on-write, and log-structured volume. But it's very hard to retrofit existing filesystems with new consistency mechanisms. Data journalling is available on ext4, but is very slow.
- Rebuilding the Merkle tree after every write, which would be extremely inefficient. Alternatively, a different authenticated dictionary structure such as an "authenticated skiplist" could be used. However, this would be far more complex.

Compare it to dm-verity vs. dm-integrity. dm-verity is very simple: the kernel just verifies read-only data against a read-only Merkle tree. In contrast, dm-integrity supports writes but is slow, is much more complex, and doesn't actually support full-device authentication since it authenticates each sector independently, i.e. there is no "root hash". It doesn't really make sense for the same device-mapper target to support these two very different cases; the same applies to fs-verity.

Q: Since verity files are immutable, why isn't the immutable bit set?

A: The existing "immutable" bit (FS_IMMUTABLE_FL) already has a specific set of semantics which not only make the file contents read-only, but also prevent the file from being deleted, renamed, linked to, or having its owner or mode changed. These extra properties are unwanted for fs-verity, so reusing the immutable bit isn't appropriate.

Q: Why does the API use ioctls instead of setxattr() and getxattr()?

A: Abusing the xattr interface for basically arbitrary syscalls is heavily frowned upon by most of the Linux filesystem developers. An xattr should really just be an xattr on-disk, not an API to e.g. magically trigger construction of a Merkle tree.

Q: Does fs-verity support remote filesystems?

A: Only ext4 and f2fs support is implemented currently, but in principle any filesystem that can store per-file verity metadata can support fs-verity, regardless of whether it's local or remote. Some filesystems may have fewer options of where to store the verity metadata; one possibility is to store it past the end of the file and "hide" it from userspace by manipulating i_size. The data verification functions provided by `fs/verity/` also assume that the filesystem uses the Linux pagecache, but both local and remote filesystems normally do so.

Q: Why is anything filesystem-specific at all? Shouldn't fs-verity be implemented entirely at the VFS level?

A: There are many reasons why this is not possible or would be very difficult, including the following:

- To prevent bypassing verification, pages must not be marked Uptodate until they've been verified. Currently, each filesystem is responsible for marking pages Uptodate via `->readahead()`. Therefore, currently it's not possible for the VFS to do the verification on its own. Changing this would require significant changes to the VFS and all filesystems.
- It would require defining a filesystem-independent way to store the verity metadata. Extended attributes don't work for this because (a) the Merkle tree may be gigabytes, but many filesystems assume that all xattrs fit into a single 4K filesystem block, and (b) ext4 and f2fs encryption doesn't encrypt xattrs, yet the Merkle tree *must* be encrypted when the file contents are, because it stores hashes of the plaintext file contents.

  So the verity metadata would have to be stored in an actual file. Using a separate file would be very ugly, since the metadata is fundamentally part of the file to be protected, and it could cause problems where users could delete the real file but not the metadata file or vice versa. On the other hand, having it be in the same file would break applications unless filesystems' notion of i_size were divorced from the VFS's, which would be complex and require changes to all filesystems.
- It's desirable that FS_IOC_ENABLE_VERITY uses the filesystem's transaction mechanism so that either the file ends up with verity enabled, or no changes were made. Allowing intermediate states to occur after a crash may cause problems.