



DFRWS APAC 2026 - Selected Papers from the 6th Annual Digital Forensics Research Conference APAC

Active block hashing for efficient integrity verification of modern storage media and their forensic duplicates^{*}

Jan Gruber^a, Felix Freiling^{b,*}, Simson L. Garfinkel^c

^a KASTEL Security Research Labs, Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, 76131, Germany

^b Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, Erlangen, 91058, Germany

^c BasisTech LLC, 1070 Broadway, Somerville, MA, 02144-2078, United States

ARTICLE INFO

Keywords:

Digital evidence
Digital investigations
File systems
Forensic tools
Evidential integrity

ABSTRACT

Verifying evidential integrity through hashing underpins the chain of custody in digital forensics, yet modern storage technologies create complications because devices no longer reliably retain stale data. Sequential bitwise hashing is computationally expensive and sensitive to block-level changes caused by TRIM/Deallocate/Unmap operations on flash-based storage, where the controller silently alters blocks once they are deallocated; file hashing avoids this but is incomplete since it excludes relevant file system (FS) metadata. This article introduces *tree-based active block hashing*, a deterministic hashing scheme that leverages file system semantics to protect the integrity of active blocks. These data blocks are organized into a Merkle tree that remains invariant to TRIM-induced changes and, rather than merely signaling that verification failed, localizes modifications, including those affecting file system metadata, enabling examiners to pinpoint affected blocks, assess their evidential impact, and perform partial integrity verification. We formalize the construction, define activation functions for FAT, NTFS, and ext file systems, and present an open-source implementation. An empirical evaluation on solid-state drives with varying file systems and fill levels shows that active block hashing outperforms sequential bitwise hashing on partially filled media and remains competitive with FS parser-mediated file hashing while offering broader integrity coverage. Limitations arise for highly filled media and in the presence of file system corruption, indicating that the approach complements existing modification detection methods, as it characterizes a trade-off between integrity protection, computational efficiency, and granularity of integrity verification.

1. Introduction

Digital forensic practitioners face growing data volumes and storage capacities. In response, the research community has proposed triage techniques (Roussev et al., 2013) and emphasized selective data acquisition (Turner, 2005; Stüttgen et al., 2013) to reduce investigative effort. However, these approaches address data volume but do not reduce the cost of integrity verification, nor do they tackle integrity challenges on solid-state drives (SSDs) that impede evidence authentication.

1.1. Motivation: TRIM, Deallocate, and Unmap

Prior forensic research has examined the implications of the

autonomous behavior of SSD controllers following the issuance of ATA TRIM, NVMe Deallocate, or SCSI Unmap commands (Bell and Boddington, 2010; King and Vidas, 2011; Nisbet and Jacob, 2019; Hadi et al., 2021). These commands, typically issued by the operating system to the storage subsystem after a file is deleted or at fixed intervals, allow the storage device to proactively erase the physical storage cells associated with the blocks, which is primarily required to enable their reuse but also to maintain optimal write speeds and prevent premature wear.

Because the execution of these commands can be queued and deferred by the storage controller in a manner invisible to the operating system, “the hash of an SSD may not be ‘stable’ over time, as obsolete data may be automatically wiped by internal optimizations” (Bonetti et al., 2013, p. 269). The challenge of documenting and verifying

^{*} This article is part of a special issue entitled: “DFRWS APAC 2026” published in Forensic Science International: Digital Investigation.

^{*} Corresponding author.

E-mail addresses: jan.gruber@kit.edu (J. Gruber), felix.freiling@fau.de (F. Freiling), simsong@acm.org (S.L. Garfinkel).

evidential integrity under these conditions has been highlighted by various researchers (Bell and Boddington, 2010; Nisbet and Jacob, 2019). In particular, Neyaz et al. (2019) demonstrated changes in hash values resulting from TRIM commands. Ultimately, this behavior undermines traditional integrity verification methods based on computing a bit-by-bit hash over all blocks of the physical disk—an observation that has also gained attention among practitioners.¹

It is important to note that even if the SSD controller supports the deterministic read after TRIM (DRAT) or deterministic zeros after TRIM (DZAT) features—which ensure that readouts following a TRIM command deterministically return zeros or a fixed pattern—the original data are irretrievable at the standard access level (except through specialized factory-access methods); the returned value is an artifact of the controller, not evidence—it lacks evidential relevance in the formal sense (Gruber and Humml, 2023). Hashing devices using DRAT or DZAT produces a stable result, but the values returned for “trimmed” blocks are evidentially meaningless. Consequently, forensic handling of SSDs necessitates a re-evaluation of hashing procedures to attain invariance to TRIM and improve efficiency.

1.2. Contributions

We formalize *activation functions* that partition every block of a file system (FS) into deterministic and non-deterministic regions, with instantiations for FAT, NTFS, and ext. Building on these, we propose *tree-based active block hashing* for evidential integrity verification of forensic images. The resulting digests (a) enable the localization of modifications and partial integrity verification through their Merkle tree (MT) structure—a feature not provided by any established forensic hashing procedure; (b) cover all relevant file system data structures, in contrast to approaches that hash only file contents and omit metadata; (c) are invariant to the execution of TRIM, Deallocate, and Unmap commands; and (d) can be computed faster than bit-by-bit hashes on partially filled media. We present three operational modes: a sequential hash over active blocks (linear), a compact MT over active blocks (CMT), and a structure-preserving variant that pads inactive regions to keep leaf positions stable under block (de-)allocation (SPMT). Finally, we evaluate an open-source implementation of the scheme, *ahsum*, on physical SSDs across three file systems, showing that active block hashing in linear and CMT modes outperforms FS driver-mediated file hashing on non-empty media and remains faster than bitwise hashing at fill levels up to approximately 91–97 %, depending on the FS, and matches FS parser-mediated file hashing while protecting a larger portion of metadata.

2. Background and related work

TRIM-induced changes affect storage media, undermining bitwise integrity verification (Bell and Boddington, 2010; Nisbet and Jacob, 2019; Neyaz et al., 2019). We review existing approaches to this challenge.

Mitchell et al. (2016) addressed the challenge of hash mismatches caused by the TRIM command on SSDs during forensic acquisitions. They proposed a method called “Deconstruct and Preserve,” which involves saving key metadata (the master boot record (MBR) or volume boot record (VBR) in their FAT-based experiments), and then overwriting that data with zeros prior to issuing the TRIM command (Mitchell et al., 2016, 2018). The authors stated that DaP is “prohibiting the activation of the garbage collector and the non-execution of TRIM commands on the SSD” (Mitchell et al., 2016, p. 4), so it does not mitigate the progressive data loss over time caused by TRIM commands issued prior to starting this process (Nisbet et al., 2013; Nisbet and Jacob, 2019).

This method involves explicitly writing to the evidence disk, which is likely impermissible under many forensic protocols. Additionally, the TRIM command might be blocked by a write-blocker, preventing proper operation of the DaP software.

Grier and Richard III (2015) introduced a framework for selective forensic imaging designed to reduce acquisition time and storage overhead while preserving data identified as relevant. A core component of their methodology is the concept of *sifting collectors*—code modules that evaluate individual blocks or contiguous regions (termed “grains”) to determine their inclusion in the final forensic container. However, the collectors are used for evidence collection, not integrity verification, which remains unaddressed in their work.

For the purpose of integrity validation, MTs have recently been used by Han et al. (2024) for the construction of digital evidence bags. In their work, a logical evidence container is built on top of a public archive file format, in which multiple digital evidence items are stored alongside their associated metadata. The hash values of the individual evidence items are organized into an MT, which cryptographically binds all items together and enables integrity verification of the entire container as well as of arbitrary subsets thereof (Han et al., 2024). Importantly, however, this work does not explore the use of MTs for integrity protection at the level of a single evidence item, such as an individual disk image or partition, which is the focus of our work.

3. Understanding active data

Our approach relies on distinguishing between data that are currently meaningful to the file system and data that the file system treats as obsolete or irrelevant. Therefore, before defining a hashing scheme, we first need to establish an abstraction of what constitutes active data within a file system. This abstraction then needs to be instantiated for concrete file systems.

3.1. A definition of activation status

Active parts of a file system may comprise both essential and non-essential data (Carrier, 2005, p. 176), since critical metadata structures as well as data units (DUs) filled with file content must be considered. Hence, data of all categories in the reference model set out by Carrier (2005, pp. 177 ff.) will be factored into the hash. The smallest parts of an FS are commonly referred to as DUs.

Definition 3.1. (Data Unit; Carrier, 2005, p. 174) A *data unit (DU)* is a “standard-sized container” constituting the smallest addressable chunk of data, i.e., clusters or blocks, which are addressed by a logical FS address.

Definition 3.2. (Active DU) A DU is *active* if it contains file system metadata or if it is explicitly marked as allocated in a metadata structure of the file system.

Management data comprise DUs of the file system, file name, metadata, and application categories. DUs explicitly marked as allocated are generally part of the content category (and in some cases part of the application category). Which DUs occupy which category of data at which location within the FS depends on the particular file system, and thus the concrete classification of activation status is necessarily file-system-specific.

3.2. Formal description of active data units

We formalize the input data, the determination of activation status, and the actual hashing.

3.2.1. Definition of the data

Let a DU be a bit string of fixed length, i.e., $\{0,1\}^b$ for some $b \in \mathbb{N}^+$. Define the set of all DUs of length b as $\mathbb{D}_b = \{0,1\}^b$. An FS is modeled as

¹ See <https://blog.elcomsoft.com/2025/06/what-trim-drat-and-dzat-really-mean-for-ssd-forensics/>.

an array of DUs of length $\kappa \in \mathbb{N}^+$, which is a finite, ordered sequence over $\mathbb{D}_b$, where each data unit is of size b , i.e.,

$$D_b^\kappa = \langle d_0, d_1, \dots, d_{\kappa-1} \rangle \quad \text{with each } d_i \in \mathbb{D}_b, \quad 0 \leq i < \kappa.$$

The set of all such arrays of arbitrary length but fixed DU size b is:

$$\mathbb{D}_b^* = \bigcup_{\kappa \in \mathbb{N}^+} (\mathbb{D}_b)^\kappa.$$

We write $\mathbb{D}^*$ for the set of all such arrays of arbitrary length and arbitrary unit size.

3.2.2. Determination of activation status

Let $\mathcal{F}$ be the set of possible FS types and let $\tau \in \mathcal{F}$. We define a file-system-specific predicate

$$\text{active}_\tau : \mathbb{D}^* \times \mathbb{N} \rightarrow \{\top, \perp\}.$$

This predicate determines whether the DU at index i in the array D_b^κ is active. Given an input array

$$D_b^\kappa = \langle d_0, d_1, \dots, d_{\kappa-1} \rangle, \quad d_i \in \mathbb{D}_b,$$

we define the function $A_\tau : \mathbb{D}^* \rightarrow (\mathbb{N} \times \{0, 1\}^*)^*$ to return the sequence of active units in the concrete instance of the file system τ , by

$$A_\tau(D_b^\kappa) = \langle (i, d_i) \mid 0 \leq i < \kappa, \text{active}_\tau(D_b^\kappa, i) = \top \rangle,$$

where the resulting tuples are ordered by their logical address i .

3.3. Case studies of active data units

We define the file-system-specific semantics of function A_τ in three case studies.

3.3.1. Activation status for FAT file systems

For FAT file systems, we consider the DUs housing the following structures as active. Since the boot sector, which is part of the FS category according to the reference model by Carrier (2005), contains information required to read the FS correctly, it must be considered active. The reserved area with FS information sector FSINFO and the backup boot sector must be considered active since they are (at least partly) occupied with data of the FS category. The file allocation table area, which can be assigned to the metadata category, is also considered active, since it is used to keep track of which data units are allocated and belong to which file. For FAT12 and FAT16, the root directory, located immediately at the beginning of the data area, is also considered active (commonly 32 sectors, when BPB_RootEntCnt is 512). For FAT32, the root directory is located at a variable position in the data area (as specified by BPB_RootClus) and the allocation of the underlying DUs is managed by the file allocation table. For the data area, the activation status of a DU, which is assigned to the content category, has to be determined by the allocation specified in the file allocation table.

3.3.2. Activation status for ext{2,3,4} file systems

FSs of the ext{2,3,4} family employ various complex structures, whose activation status needs to be determined. The primary superblock, located at offset 1024 and occupying 1024 bytes, resides at the very beginning of the file system and is replicated as backup copies across block groups. Since it stores essential information such as block size, the number of blocks per block group, the offset of the first data block, and bookkeeping data about free inodes and blocks, the superblock and its backups must be considered active.

Similarly, the group descriptor table, located immediately after the superblock, is also classified as active, as it contains information about the block groups. In addition to data blocks, each block group is associated with an inode table, an inode bitmap, and a block bitmap,

although the flex_bg feature may physically place these in another group; backup copies of the superblock and the group descriptor table reside only in selected block groups when the default sparse_super feature is enabled. The block bitmaps can be consulted to determine the activation status of each block within a block group. The aforementioned metadata structures “are all allocated data even though they are not allocated to a file” (Carrier, 2005, p. 410). Data blocks are also marked according to their allocation status in the corresponding block group’s block bitmap. Journal blocks are allocated in the same way as regular blocks and are assigned to the special inode 8.

An interesting detail concerns the “reserved sectors at the beginning of the file system,” for which “it is less clear what their allocation status should be because there is no bitmap entry for them” (Carrier, 2005, p. 410). For instance, The Sleuth Kit considers all reserved blocks up to and including the superblock as allocated; beyond that, the block bitmap is consulted.

3.3.3. Activation status for the NTFS file system

Although NTFS is a complex FS, deriving activation status is straightforward, since “everything is a file,” including administrative data (Carrier, 2005, p. 274). The allocation of each cluster is tracked in the FS metadata file \$Bitmap; consulting its \$DATA attribute, which contains one bit per cluster, is sufficient to determine activation status for any block in the FS (Carrier, 2005, p. 311).

4. Active block hashing

4.1. The general idea

We aim to verify evidential integrity more efficiently and effectively. We pursue this through two key ideas reflected in the deterministic, hash-based binding commitment procedure.

First, integrity verification is optimized by hashing only a reduced subset of the data while still covering all relevant portions and remaining invariant to changes induced by TRIM/Deallocate. To this end, we consider only active DUs, as defined in Section 3.

Second, we use a hierarchical hashing scheme that enables the localization of potential data modifications. Specifically, we construct an MT (Merkle, 1979, 1989), in which each leaf node represents the hash of a group of active data blocks, and each internal node is the hash of its children. This process yields a single Merkle root, which serves as a cryptographic commitment to the entire data set and allows for efficient modification detection.

This method does not discard evidence; it excludes blocks whose content the drive has already invalidated—solving evidential problems with SSDs while reducing runtime.

4.2. Security requirements

4.2.1. Adversarial assumptions

We assume an adversary with full post-acquisition write access to the raw storage medium (i.e., at the block level, not through the file system driver), but without the ability to violate standard cryptographic hash assumptions or to influence the deterministic activation function or the verification procedure. Under these assumptions, any modification affecting active DUs must be detectable, whereas changes confined to inactive DUs are considered out of scope by design.

4.2.2. Goal and requirements

The proposed construction aims to achieve second-preimage resistance and collision resistance (Menezes et al., 1996, p. 323 f.) with respect to a logical acquisition state S , defined by the active DUs, their ordering, and their content. Since the construction, unlike conventional bitwise hashing, selectively commits to active data based on file system semantics, additional requirements are necessary to preserve these cryptographic properties. Accordingly, the procedure must satisfy the

following requirements: it must

- bind the complete logical acquisition state to a single root hash (*binding to logical acquisition state*);
- bind each active DU to its logical position to prevent reordering or relocation (*positional binding*);
- ensure that the aggregation structure is unambiguous such that distinct tree structures cannot yield the same root hash (*structural soundness*);
- allow any verifier to deterministically reconstruct the identical tree and root hash from the serialized state and public parameters (*deterministic verifiability*).

Under these conditions, the root hash constitutes a binding integrity commitment to the logical acquisition state. Once published, any modification to the active DUs, their ordering, or their activation status results in a distinct root hash, except with negligible probability (Appendix B). Accordingly, under standard cryptographic assumptions, the root hash functions as a modification detection code for a semantic representation of the storage medium.

4.3. Hash tree calculation

The active DUs are used to construct an MT (Merkle, 1979, 1989). The parameters $\eta \geq 1$ and $k \geq 2$ govern the granularity and structure of the construction: η controls how many DUs are aggregated into a single leaf, while k determines the arity of the resulting tree.

4.3.1. Grouping of active data units

Given the ordered sequence of active DUs $A_\tau(D_b^k) = \langle a_0, a_1, \dots, a_m \rangle$ and a grouping parameter $\eta \in \mathbb{N}^+$, we construct a sequence of groups:

$$G = \langle g_0, g_1, \dots, g_{\lceil (m+1)/\eta \rceil - 1} \rangle.$$

Each group g_x is defined as the ordered sequence of up to η active units:

$$g_x = \langle a_j \rangle_{j=x\eta}^{\min((x+1)\eta-1, m)}.$$

4.3.2. Tree construction

Let $H: \{0,1\}^* \rightarrow \{0,1\}^k$ be a cryptographic hash function and $k \geq 2$ the tree arity. To ensure domain separation, we prepend a fixed tag $(\alpha, \beta \in \{0,1\}^8, \text{ pairwise distinct})$ to each hash input: α for leaf nodes and β for internal nodes. We first serialize each group g_x into a bit string $\hat{g}_x$ by concatenating the tuples (i, d) in order, where each index i is encoded at a fixed width of $\lceil \log_2 k \rceil$ bits:

$$\hat{g}_x = \parallel_{(i,d) \in g_x} (i \parallel d)$$

The leaf nodes $H_x^{(0)}$ are then computed as:

$$H_x^{(0)} = H(\alpha \parallel \hat{g}_x), \quad 0 \leq x < |G|$$

At every level $\ell \geq 1$, we use a distinct domain tag β to identify the nodes at that level as internal nodes and define them recursively as follows:

$$H_j^{(\ell)} = H\left(\beta \parallel H_{kj}^{(\ell-1)} \parallel \dots \parallel H_{\min(kj+k-1, |H^{(\ell-1)}|-1)}^{(\ell-1)}\right),$$

$$\text{for } 0 \leq j < \left\lceil \frac{|H^{(\ell-1)}|}{k} \right\rceil.$$

The recursion terminates at the smallest level L such that $|H^{(L)}| = 1$. The single remaining value

$$h_{p,\tau} = H_0^{(L)}$$

is taken as the root hash of partition p , which constitutes the active block hash of the FS τ .

Finally, let $h_{p_1,\tau_1}, \dots, h_{p_n,\tau_n}$ be the per-partition root hashes, ordered by ascending partition start address. These are aggregated into a global root using the same k -ary internal-node construction, except that each aggregation node prepends a distinct domain-separation tag $\delta \in \{0,1\}^8$ ($\delta \neq \alpha, \beta$) and the recursion proceeds as above until a single value remains (for a single partition, no aggregation node is added, so $h_{\text{global}} = h_{p_1,\tau_1}$). The resulting root hash h_{global} constitutes the active block hash of the complete device.

4.4. Modes of operation

We distinguish three modes of operation for active block hashing, which offer different trade-offs between verification granularity, run-time performance, and the ability to localize changes:

Linear mode:

Hashing all active DUs consecutively.

Compact Merkle tree mode:

Constructing an MT from active DUs only.

Structure-preserving Merkle tree mode:

Constructing an MT from all DUs, where inactive units are masked.

These modes are determined by the choice of the activation status function and the tree parameters.

4.4.1. Linear mode

As a special case, the active DUs may be hashed sequentially. Given $A_\tau(D_b^k) = \langle a_0, a_1, \dots, a_m \rangle$, where $a_j = (i_j, d_j)$, the linear hash is:

$$H\left(\alpha \parallel \parallel_{j=0}^m (i_j \parallel d_j)\right).$$

This corresponds to a degenerate tree construction where no leaf aggregation is performed, so its root is treated as a single leaf and carries the leaf tag α . In linear mode each index i_j is a physical address at a fixed device-wide width, and no per-partition aggregation is applied: the active DUs of all file systems are concatenated in ascending physical-address order into a single hash. Consequently, it does not support the localization of modifications, but may be useful when efficiency is prioritized and the localization of changes is not required.

4.4.2. Compact MT mode

The compact Merkle tree (CMT) mode constructs an MT using only the sequence of active data units returned by $A_\tau(D_b^k)$. It allows for the localization of changes within active DUs, provided that the *sequence* of active DUs (i.e., the activation status of the file system) remains constant. However, if the activation status changes (e.g., a file is deleted), the sequence of active DUs changes, causing a “shift” in the assignment of DUs to tree leaves. This shift propagates through the tree, causing the root hash and multiple intermediate nodes to differ, thereby hindering precise localization.

4.4.3. Structure-preserving MT mode

To enable the localization of changes even when the activation status of individual DUs changes, while retaining invariance to TRIM for inactive areas, we introduce the structure-preserving Merkle tree (SPMT) mode. This mode relies on a variant of the activation function, $A'_\tau: \mathbb{D}^* \rightarrow (\mathbb{N} \times (\{0,1\}^* \cup \{\perp\}))^*$, which preserves the positions of inactive DUs by masking them with the symbol $\perp$:

$$A'_\tau(D_b^k) = \langle (i, d'_i) \rangle_{i=0}^{k-1}$$

where

$$d'_i = \begin{cases} d_i & \text{if active}_\tau(D_b^k, i) = \top \\ \perp & \text{otherwise} \end{cases}$$

Including these masking values keeps the topology of the tree stable

regardless of activation changes. Here, leaves group all κ positions of $A_\tau(D_b^\kappa)$ rather than the active subsequence:

$$g'_x = \langle (i, d'_i)_{i=x\eta}^{\min((x+1)\eta-1, \kappa-1)} \rangle \quad \text{for } 0 \leq x < \lceil \kappa / \eta \rceil.$$

The leaf hash for a group g'_x in SPMT mode is computed as follows: if the group contains at least one active DU, the leaf is $H(\alpha \parallel \tilde{g}'_x)$, where $\tilde{g}'_x$ serializes only the active units (i, d'_i) within the group (excluding inactive ones, i.e., those with $d'_i = \perp$). If all DUs in the group are inactive, a distinct domain tag $\gamma \in \{0,1\}^8$ ($\gamma \neq \alpha, \beta, \delta$) is used and the leaf is the domain-separated constant $H(\gamma)$, marking the group as inactive without hashing any data. Since each active tuple carries its absolute block address i , the serialization is inherently position-bound without requiring an explicit position prefix. Any activation-status change is detectable because $\alpha \neq \gamma$. The construction allows for precise localization of content modifications in active DUs, at the cost of processing a larger number of leaves (equal to the total number of groups).

Fig. 1 illustrates the construction of a 2-ary tree, where two DUs are grouped to form a leaf node. This demonstrates the two tree-based modes of operation: comparing Fig. 1a with Fig. 1b and c shows that changes to activation status can only be localized when inactive data units are represented by masking values $\perp$, i.e., using $A_\tau(D_b^\kappa)$. This means, however, that the number of leaf nodes will be constant with respect to activation-status changes, and hence the total number of nodes in the tree will be constant as well.

4.5. Computational complexity of the modes

Referring to the notation introduced in Section 3.2, a conventional bitwise hashing approach processes all blocks in D_b^κ , resulting in a computational cost that is linear in the total number of blocks, i.e., $\mathcal{O}(|D_b^\kappa|)$ updates of the hashing context.

4.5.1. Linear mode

In contrast, linear active block hashing operates only on the subset of active DUs, denoted by $A_\tau(D_b^\kappa)$. Only these blocks are fed into the hashing context, yielding a computational complexity of

$$\mathcal{O}(|A_\tau(D_b^\kappa)|),$$

which can be substantially smaller than $|D_b^\kappa|$, depending on the fill level of the storage medium. As this approach produces a single hash value and does not maintain auxiliary data structures, the requirements to store the result are negligible.

4.5.2. CMT mode

In the CMT mode, active DUs are first grouped using a grouping parameter $\eta \in \mathbb{N}^+$, resulting in

$$l = \left\lceil \frac{|A_\tau(D_b^\kappa)|}{\eta} \right\rceil$$

groups of DUs. A hash tree with branching factor $k \geq 2$ is then constructed over these groups. In this setting, l leaf and $\mathcal{O}\left(\frac{l}{k-1}\right)$ internal-node hashing operations are required. Thus, the overall computational complexity remains linear in the number of groups, while integrity verification of a single group can be performed in $\mathcal{O}(\log_k l)$ time. The storage complexity is likewise linear in the number of leaves.

4.5.3. SPMT mode

SPMT active block hashing augments this approach by explicitly retaining the positional structure of the original data. All blocks, both active and inactive, are grouped into

$$l' = \left\lceil \frac{|D_b^\kappa|}{\eta} \right\rceil$$

leaves. However, fully inactive groups reduce to a fixed padding constant $H(\gamma)$, so no hash over the group's contents needs to be computed. As described in Section 4.4, the cost of forming a leaf from a group containing at least one active DU is proportional to the number of active units in the group. Fully inactive groups reduce to padding leaves, whose input has constant size regardless of η . Consequently, the leaf-level cost is dominated by groups that contain active DUs; in the best case (active DUs contiguous), this matches the CMT leaf count $\lceil |A_\tau(D_b^\kappa)| / \eta \rceil$. In all cases, $\mathcal{O}\left(\frac{l'}{k-1}\right)$ internal-node hashing operations are required. The tree structure is preserved regardless, so the storage remains proportional to the total number of leaves l' .

5. Implementation

We implemented a prototype for computing both tree-based and linear active block hashes in C++ as a standalone program.² The implementation relies on `libtsk` for partition and file system parsing and on `libcrypto` for cryptographic hash computation via its EVP interface.

The program takes as input either a disk image (potentially segmented into multiple files) or a block device path. To compute an active block hash value for a storage medium, the partitioning information is first parsed to identify the partitions present, using schemes such as GUID Partition Table (GPT) or MBR. If no partitioning information is available, the medium is treated as a single partition. For each partition, the contained FS τ is identified and traversed to determine all active blocks containing file content or file system metadata, which corresponds to applying the respective activation function A_τ . This procedure is performed independently for each detected FS. Depending on the selected mode of operation (Section 4.4), the active DUs are either processed sequentially in a single hashing context to produce a linear active block hash value, or aggregated per partition into fixed-size groups and hashed to form the leaf nodes of an MT, which is recursively aggregated in a k -ary fashion until a single root hash is obtained. The leaf hashes commit both to their logical addresses in the FSs and to the data bytes themselves. The tree is not perfect, since internal nodes are not padded, and each internal hash commits only to the ordered sequence of its existing child hashes. Because child hashes are of fixed length λ , the number of children is unambiguously encoded in the hash input length, making explicit padding unnecessary for structural soundness. Combined with fixed arity, deterministic child ordering, and domain separation, the construction is unambiguous. Domain separation is enforced by prepending distinct, fixed tags to the hash inputs for leaf nodes, internal nodes, masking values, and partition-root aggregation nodes, preventing structural ambiguity and cross-domain hash collisions. When multiple partitions are present, the per-partition root hashes of the tree-based modes are combined under the aggregation tag in a deterministic order; in linear mode, the single hashing context spans all partitions.

For tree-based modes, the resulting MT is serialized and output in JSON format (Bray, 2017), which provides a lightweight and human-readable representation that enables transparent inspection and tool-independent processing. Only the leaf commitments and a single root hash as a compact commitment to the entire image are stored; the tree structure is left implicit.

² See <https://github.com/jgru/active-block-hashing>.

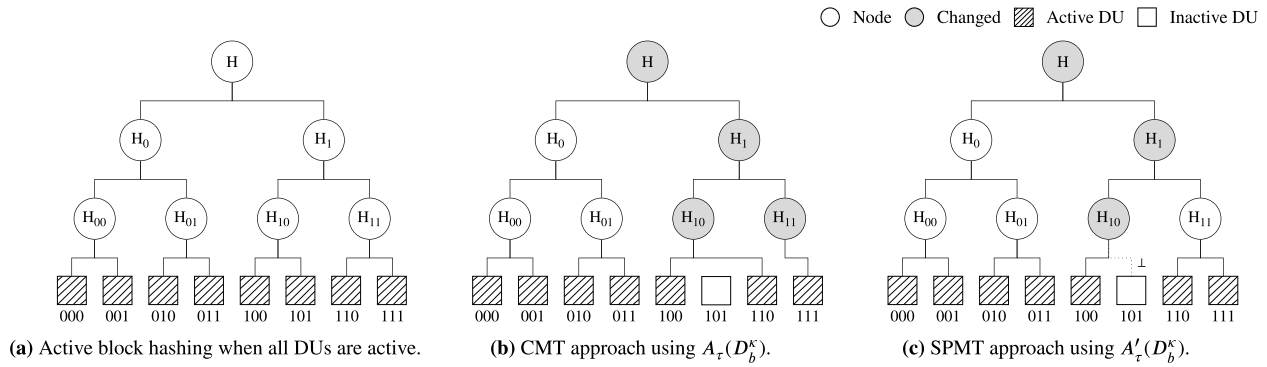


Fig. 1. Tree structures (with $k=2$ and $\eta=2$). The comparison of modes highlights the impact of the construction strategy. (a) Baseline with all DUs active. (b) CMT mode: deallocation of a unit shifts the leaf assignment, so localization is reliable only while the set of active units is constant. (c) SPMT mode: inactive units are masked with $\perp$, preserving the topology even when units become inactive.

6. Evaluation

To evaluate our proposed method and compare it against alternatives, we first tested its technical correctness. After establishing correctness of the proposed procedure, we compared its runtimes against alternative ways of checking evidential integrity. All experiments were conducted using reproducible, literate-programming-based workflows orchestrated with GNU `make`. The complete documentation and results are available with our prototype implementation.

6.1. General setup of the testbed

Hardware and operating system. The experiments with the forensic hashing procedures were conducted on an Intel NUC 10 with an Intel Core i7-10710U processor and 32 GB DDR4 RAM. Storage media were attached via SATA III through a dedicated flex cable for 2.5-inch drives. The host system ran Debian GNU/Linux 13 (Codename: Trixie) with the 6.12.35 Linux kernel on a 64-bit architecture.

Measurement software. For measuring wall time (i.e., elapsed real time) of our proposed procedure and existing approaches, we employed the benchmarking tool `hyperfine` in version 1.19.0.³ It executes arbitrary shell commands and performs statistical analysis across multiple runs, reporting metrics such as mean runtime and standard deviation, while offering the ability to conduct warmup runs and run cache-clearing commands prior to the measurements.

Comparison tools. We compared the runtime of our tool `ahsum` with both bit-by-bit and file hashing approaches using the most widely adopted tools available at the time of writing. For bit-by-bit hashing, we used `sha256sum` from the GNU `coreutils` package. For file hashing, we evaluated two representative methods. First, we considered *FS driver-mediated file hashing*, as implemented in Kornblum’s `hashdeep` (Altheide and Carvey, 2011, p. 57), which uses the user-space file API to read and hash file contents. Second, we evaluated *FS parser-mediated file hashing*, as implemented by `fiwalk` (short for “file and inode walk”) by Garfinkel (2009), which accesses file contents directly by parsing FS metadata structures and hashes them at the block level rather than through the operating system interface.⁴

During our experiments, we encountered issues with Sleuth Kit’s automatic FS detection, which both `fiwalk` and `ahsum` rely upon. These problems were traced to the YAFFS2 detection logic, which either failed to properly distinguish YAFFS2 from ext FSs or required excessive time

Table 1

Tools used for comparing the runtime of integrity checks.

Tool	Version	Type
ahsum (this work)	0.0.1	active
fiwalk (sleuthkit)	26ea7e4	file-based
sha256deep (hashdeep)	4.4	file-based
sha256sum (GNU coreutils)	9.7	bit-by-bit

to do so. To address this, we patched Sleuth Kit to exclude the rarely encountered YAFFS2 FS from automatic detection. The tools, their respective versions, and their modes of operation are summarized in Table 1.

6.2. Technical correctness

6.2.1. Invariance to TRIM-induced changes

Our evaluation of technical correctness proceeds in two steps. First, a repeated randomized test suite confirms invariance to *simulated* TRIM on loop-mounted images; second, a confirmatory run tests it against real TRIM from the SATA command set on a physical SSD (Intenso 128 GB SSD SATA-600).

Each randomized trial creates a fresh ext4 FS, directly on the image or in a GPT partition, populated with a random number of randomly sized files of pseudo-random content, and then deletes a random, non-empty proper subset with `rm -f`. The deleted files’ extents (identified via `debugfs`) are overwritten with zeros in place on the unmounted device, emulating TRIM; as only unallocated *data* blocks change, any difference in the active hash would indicate a genuine defect rather than a file system artifact. Before and after zeroing, we compute a bit-by-bit hash with `sha256sum` and, via `ahsum`, the linear active-block hash and both MT variants (compact and structure-preserving); a trial passes if and only if the SHA-256 hash changes while all three active hashes are unchanged. All choices—file count, sizes, content, and the deleted fraction (1–99%)—are randomized and derived from a single reported seed. Across $N=200$ trials, spanning 16–1024 files on a 512 MiB image and a total of 8.56 GiB of zeroed unallocated space, the SHA-256 hash changed in every trial while all three active hashes remained invariant (200/200). The physical SSD confirmed this: after `fstrim`, the SHA-256 hash changed while the active-block hash produced by `ahsum` did not.

6.2.2. Sensitivity to and localization of modifications

To assess the sensitivity of the proposed method to modifications of active data blocks, we manually altered the content of a single active block and recomputed the active block hash using all three modes. In all cases, the resulting (root) hash values changed, confirming sensitivity to modifications of relevant data (formally argued in Appendix B). To assess localization capabilities, we compared the MTs generated before

³ See <https://github.com/sharkdp/hyperfine>.

⁴ For this study, we extended `fiwalk` to support SHA-2 family hashing functions. This patch has been merged upstream in commit 26ea7e4.

and after the modification. For both the CMT and SPMT modes, this comparison enabled us to precisely localize the modified block.

In addition, we performed an experiment involving file deletion on an ext4 file system. As expected, the active block hashes changed accordingly. Only with the structure-preserving method did inspection of the corresponding MTs allow us to trace the affected blocks, which were found to span two block groups. Further changes were observed at additional block locations, most likely corresponding to file system metadata structures such as block bitmaps, inode bitmaps, inode tables, and backup superblocks.

6.3. Runtime measurement with SSDs

To compare the runtime of the different hashing methods, we used two consumer SSDs with a capacity of 128 GB connected via SATA.⁵ SATA interfaces remain widely deployed in existing forensic casework, and this capacity is sufficient to derive general performance estimates while still being small enough to allow a large number of experiments to be conducted within a reasonable time frame.

6.3.1. Experimental design

To systematically assess the impact of partition count, fill level, and FS type, we designed 66 experimental configurations varying the following parameters on two equal-sized SSDs: partition count (single vs. four partitions), file system (ext4, FAT32, NTFS), and fill level in 10 % increments. The setup was automated via a custom Bash library handling GPT partitioning, FS creation, and population with random files sized between 32 KiB and 512 MiB within a four-level directory hierarchy.

After completion of the setup in a particular configuration, we ran `ahsum` in all three modes, `fiwalk`, `sha256deep`, and `sha256sum` (more details in Table 1) on cold caches three times on each of the two SSDs using `hyperfine`, averaging the resulting measurements to determine the respective runtime. `hyperfine` was pinned to a dedicated CPU core, and disk caches were cleared before each run. Again, the GNU `make` program was used to manage these factorial combinations automatically and to execute measurements reproducibly.

6.3.2. Results

The full per-tool runtimes across all fill levels and partition counts are listed in Tables 5–7 in Appendix A; we summarize the key findings below.

Fig. 2 illustrates that average runtime increases approximately linearly with the fill level for all methods processing exclusively allocated data, albeit with varying slopes. As expected, the runtime of bit-by-bit hashing with `sha256sum` is independent of the fill level and remains constant. It constitutes the baseline, and its constant runtime confirms a raw sequential read. In contrast, FS driver-mediated file hashing using `sha256deep` exhibits the steepest increase in runtime and becomes less efficient than bit-by-bit hashing once the fill level exceeds approximately 53 % on FAT32 and approximately 54 % on ext4 and NTFS.⁶

6.3.2.1. Comparison to `fiwalk`. We consider FS parser-mediated file hashing, as implemented by `fiwalk`, as the closest competitor to active block hashing; hence, we relate the runtimes of active block hashing in its various modes to this tool.⁷ Across all three file systems, `ahsum` in linear and CMT modes exhibits a common pattern: overhead relative to `fiwalk` is largest at low fill levels and diminishes as fill increases. For FAT32, `ahsum` is—except for empty file systems—consistently faster,

with average improvements of approximately 6 % (CMT) and 7 % (linear) across both partition configurations. For NTFS, `ahsum` in CMT and linear modes is approximately 3 % slower at 10 % fill in single-partition and approximately 4 % in four-partition configurations. The overhead falls below 1 % at ≥ 80 % fill in single-partition setups and vanishes at full capacity, with a maximal improvement of approximately 0.7 % in linear mode. In four-partition configurations, `fiwalk` remains slightly faster throughout. For ext4, the overhead is more substantial: on empty file systems, `ahsum` is orders of magnitude slower than `fiwalk`, because it hashes per-block-group metadata (backup superblocks, bitmaps, inode tables; see Section 3.3.2) that `fiwalk` does not process, even when a block group contains little or no file data—resulting in additional overhead at low fill levels. At 10 % fill the overhead is still approximately 23 % and decreases steadily thereafter. In linear mode, it falls below 1 % at ≥ 70 % fill for both partition configurations; in CMT mode, the same threshold is reached at ≥ 70 % for single-partition and ≥ 80 % for four-partition setups. Linear mode becomes faster than `fiwalk` at ≥ 80 % fill (single-partition) and ≥ 90 % (four partitions), reaching a maximal improvement of approximately 1.8 % at full capacity; CMT mode crosses to faster only at ≥ 80 % (single-partition) and at full capacity (four partitions), with a maximal improvement of approximately 1.6 %.

In SPMT mode, runtimes are substantially higher for partially filled media: at a 30 % fill level, runtime increases by approximately 165 % on FAT32 and almost 195 % on ext4 relative to `fiwalk`, whereas at 70 % fill the gap narrows to approximately 25 % and 35 %, respectively. This mode remains faster than bit-by-bit hashing up to approximately 59 % fill on FAT32, 79 % on ext4, and 82 % on NTFS.

6.3.2.2. Comparison of the modes. Table 2 reports speedups between the linear, CMT, and SPMT modes across different disk fill levels. The tree-based modes were configured with $\eta = 256$ and $k = 2$. With a 4 KiB block size, $\eta = 256$ yields a leaf granularity of 1 MiB, balancing localization precision against output compactness; $k = 2$ follows the standard binary MT convention. The results show that CMT mode performs comparably to the linear baseline (speedups consistently near 1). In contrast, SPMT exhibits substantially higher runtimes at low fill levels (up to $32.6\times$ on empty disks, approximately $1.8\times$ at 50 % fill). As disk utilization increases, all three modes converge, yielding equivalent runtimes at full capacity.

6.3.2.3. Influence of tree parameters. To quantify the influence of the tree parameters η (DU grouping) and k (arity), we swept $\eta \in \{1, 8, 64, 512\}$ and $k \in \{2, 4, 8, 16\}$ on one of the 128 GB SSDs at 25 % and 75 % fill (three runs each). Table 3 shows the results. For $\eta \geq 64$ and $k \in \{2, 4\}$, runtime is I/O-dominated and nearly constant across both parameters. For $\eta \geq 64$, increasing k to 8 or beyond mainly raises run-to-run variance, with only a modest upward drift in mean runtime; at smaller η this effect is absent. At $\eta = 1$, per-block serialization introduces noticeable overhead. Output size scales as $1/\eta$, halving with each doubling of η , and is k -independent because the serialized output contains only leaf records. SPMT mode shows the same parameter insensitivity; the dominant difference here is I/O time as it reads all FS blocks.

7. Discussion

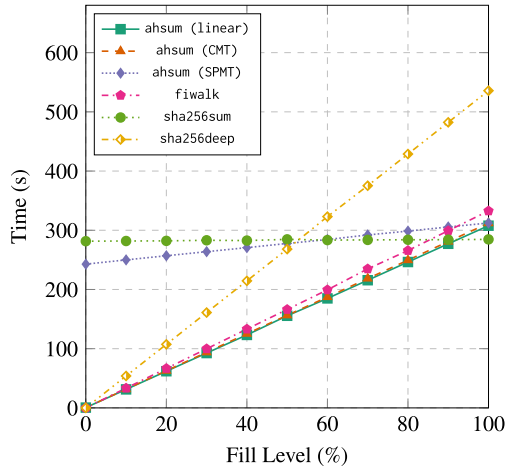
7.1. The need for active block hashing

On SSDs, deallocated blocks are either silently altered by deferred TRIM/Deallocate or deterministically overwritten under DRAT/DZAT; including such blocks in a digest thus either destabilizes re-verification or wastes effort on evidentially meaningless data. By committing only to active blocks, active block hashing avoids both problems, achieving TRIM invariance and, on partially filled media, lower runtime (Section 6), thereby addressing the reliability and efficiency demands that grow with storage volumes and investigative backlogs (Hargreaves et al., 2024, p. 8).

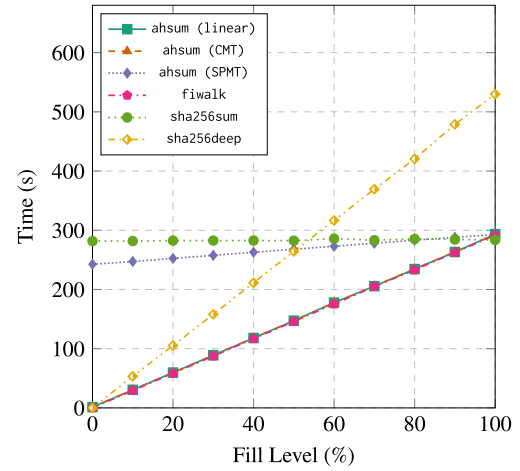
⁵ SSDs used: *Intenso Top III* and *Samsung 830*.

⁶ All crossover fill levels reported in the following are obtained by linear interpolation of the mean runtimes between adjacent measured fill levels and are averaged over both partition layouts.

⁷ Computed as $(\bar{T}_{\text{fiwalk}} - \bar{T}_{\text{ahsum}}) / \bar{T}_{\text{fiwalk}} \times 100$.



(a) Results for GPT-formatted SSDs with four partitions, each containing a FAT32 FS.



(b) Results for GPT-formatted SSDs with four partitions, each containing an NTFS FS.

Fig. 2. Mean runtime of bit-by-bit hashing, file hashing, and active block hashing as a function of device fill level, measured across two 128 GB SSDs attached via SATA III. The MT modes of *ahsum* were configured to use $\eta = 256$ for block grouping and $k = 2$ as the branching factor. *fiwalk* was configured to ignore system files and walk only allocated files.

Table 2

Speedup of the operational modes of active block hashing ($\eta = 256$ and $k = 2$) processing ext4-formatted 128 GB SSDs (averaged over devices and partition layouts). In each column “X vs. Y,” the speedup of mode Y relative to mode X is computed as $\frac{T_X}{T_Y}$; hence, values above 1 denote faster execution of mode Y.

Fill (%)	CMT vs. Lin.	SPMT vs. Lin.	SPMT vs. CMT
0	1.0031	32.6499	32.5475
10	1.0027	6.9178	6.8992
20	1.0031	3.9194	3.9071
30	1.0025	2.7736	2.7668
40	1.0024	2.1633	2.1581
50	1.0029	1.7826	1.7775
60	1.0015	1.5300	1.5276
70	1.0033	1.3450	1.3405
80	1.0028	1.2054	1.2020
90	1.0030	1.0938	1.0905
100	1.0025	1.0028	1.0003

7.2. Merits of active block hashing

Table 4 provides an overview of the data coverage offered by the considered methods for evidential integrity checks. As our runtime evaluation shows (Tables 5–7), FS driver-mediated file hashing is inefficient due to the overhead incurred by mounting the file system, parsing

Table 3

Parameter sensitivity of *ahsum* (CMT, NTFS FS on a single 128 GB SSD). All 4×4 combinations were measured; representative slices are shown.

	Runtime (s)		Output size (MiB)	
	25 %	75 %	25 %	75 %
Varying η ($k = 2$):				
1	83.9 $\pm$ 0.2	249.4 $\pm$ 0.2	962.8	2894.4
8	75.7 $\pm$ 0.5	225.0 $\pm$ 0.3	128.0	386.2
64	74.3 $\pm$ 0.1	220.8 $\pm$ 0.5	16.0	48.3
512	74.1 $\pm$ 0.2	219.7 $\pm$ 0.6	2.0	6.0
Varying k ($\eta = 64$):				
2	74.3 $\pm$ 0.1	220.8 $\pm$ 0.5	16.0	48.3
4	74.5 $\pm$ 0.3	220.7 $\pm$ 0.5	16.0	48.3
8	79.2 $\pm$ 1.1	230.5 $\pm$ 0.7	16.0	48.3
16	87.0 $\pm$ 15.0	232.9 $\pm$ 14.7	16.0	48.3

Table 4

Comparison of the integrity protection provided by the considered methods. Coverage is classified according to the categorization proposed by Carrier (2005). For each data category, it is denoted as none (○), partial (◐), or full (●).

Category	File Hashing			
	Driver-Mediated	Parser-Mediated	Active Block	Bit-by-bit
File system	○	◐	●	●
File name	●	●	●	●
Metadata	○	◐	●	●
Content	◐	◐	◐	●
Application	○	◐	●	●

its structures, and opening and reading each individual file via the user-space file API. Moreover, it is the most selective method and therefore provides the weakest integrity protection, covering only data of the file name category and allocated content data (see Table 4). Consequently, it is insufficient with respect to integrity protection for most use cases, since metadata are not covered. Bit-by-bit hashing represents the opposite extreme, as it includes all potentially existing metadata; however, below the mode- and file-system-dependent crossover points, it is less time-efficient because all unallocated blocks are also processed. FS parser-mediated file hashing is more efficient than both aforementioned approaches, as it avoids system calls for opening individual files while also excluding unused blocks from processing; nevertheless, it still protects mainly file contents, inode numbers, and MACB timestamps. Although some portions of metadata are exported by *fiwalk* to a log or DFXML file (Garfinkel, 2012) and could, in principle, be hashed separately, integrity protection remains incomplete because only a subset of metadata is covered. Whether data from the application and FS categories are included depends on (a) the FS and (b) the configuration of the tool. For instance, such coverage is possible only for NTFS, where “everything is a file,” and *fiwalk* provides options to include or exclude NTFS system files.

The proposed approach of hashing only the active blocks of an FS overcomes these limitations, as it incorporates all potentially relevant data from the metadata, file name, file system, and application categories. Beyond coverage, its MT structure localizes modifications, including within file system metadata, and supports partial integrity verification through Merkle proofs. Active block hashing targets the post-mortem workflow: an examiner images an SSD, computes a hash

over the image, and later re-verifies that hash to confirm evidential integrity. With a full-disk hash, this re-verification can fail even when no tampering has occurred—because the SSD controller may alter unallocated regions in response to deferred TRIM/Deallocate commands issued before any initial or subsequent acquisition. Active block hashing excludes those regions, so the hash remains stable across re-verification. Note that the scheme governs verification, not acquisition: acquiring the entire device remains advisable, since unallocated sectors might not yet have been trimmed or cleared by the controller and may still hold residual data at acquisition time. For traditional magnetic storage media, examiners should continue using bitwise hashing, since unallocated blocks commonly carry meaningful data on such devices. However, shingled magnetic recording (SMR) drives need further investigation, because they may honor TRIM-like discard, so their unallocated regions are not necessarily stable either.

7.3. Limitations of active block hashing

While active block hashing offers efficiency and robustness, it has limitations. At high fill levels, runtime exceeds that of bit-by-bit hashing for the chosen tree parameters: above approximately 97 % fill on ext4 and NTFS and 91 % on FAT32 in linear and CMT modes; above approximately 79 % on ext4, 82 % on NTFS, and 59 % on FAT32 in SPMT mode. These crossover fill levels are specific to the two consumer SATA III SSDs of a single 128 GB capacity used here. Because the trade-off depends on device throughput and the CPU cost of hashing, the absolute crossover points may shift on other interfaces (e.g., NVMe), capacities, or vendors; we leave a broader hardware evaluation to future work. In the tree-based modes, the MTs become large, even when we only store the leaves and the root hash. For example, in CMT mode with $\eta = 64$ and $k = 2$ at 75 % fill of the 128 GB SSD, they reach ~ 48 MiB, which may be unacceptable in some applications.

The approach also fundamentally excludes unallocated blocks, which may contain recoverable data on magnetic media. On NAND flash-based storage, unallocated blocks are generally not recoverable at the standard access level without specialized toolkits using factory access mode, which are not employed in routine forensic investigations. Even the extreme case of a volume formatted shortly before acquisition, rendering active block hashing effectively useless, seldom forfeits recoverable evidence: since Windows Vista, a full format zero-fills the entire volume, and even a quick format has been shown to leave data on TRIM-enabled drives largely unrecoverable (King and Vidas, 2011).

In principle, the formal construction of the file-system-specific activation function A_r must be carried out individually for each FS. In practice, however, this burden is limited: because our implementation builds on `libtsk`, any FS whose allocation and metadata structures it already parses is supported, as the activation function then reduces to consulting the existing allocation logic, as we did for FAT, NTFS, and ext; nevertheless, a file system that `libtsk` does not yet parse must first be added there.

Additionally, active block hashing depends on an FS that is both structurally intact and internally consistent. The activation function A_r derives the set of active blocks by parsing FS metadata, so these structures must not merely be parseable but must also carry consistent field values. If essential structures, such as allocation bitmaps or the metadata that locates them, are unreadable or inconsistent, as is common for corrupted or partially wiped partitions in forensic images, A_r cannot map the data units, and no active hash can be computed.

Finally, the MBR, primary and secondary GPT headers and partition tables are not yet incorporated into the active block hash, so a modification confined to them, leaving the active FS content untouched, goes undetected, which may matter in workflows that must attest to the partitioning scheme. Changes to file system semantics, by contrast, are reflected in the affected active block hashes. Extending the scheme to also protect the partitioning structures is straightforward, e.g., by

committing them as an additional always-active region that is aggregated into the root hash.

8. Future work

This work suggests several directions for future research beyond the scope of the present evaluation. While the current evaluation focuses on sequential hashing on commodity storage hardware, modern systems increasingly rely on parallelized I/O, including multi-queue NVMe devices and distributed storage infrastructures. We plan to investigate adaptive parallelization strategies that dynamically adjust the degree of concurrency based on observed I/O behavior, enabling the hashing process to better exploit available hardware resources without incurring unnecessary overhead. The range of target hardware might also need to be extended: SMR drives, whose host-aware, host-managed, or device-managed TRIM-like behavior blurs the classical distinction between solid-state and magnetic media, are natural candidates for evaluation.

We see potential to further improve the ability to localize changes. One option is to choose adaptive data granularity of leaves, i.e., to use a finer granularity of integrity protection for metadata structures and a coarser granularity for file content. Another option is to treat file contents and file system metadata as first-class objects. This includes integrity structures that explicitly separate data and metadata, as well as an evaluation of how such designs support forensic tasks such as selective verification, redaction, and attribution of changes. These directions frame active block hashing as a flexible foundation for integrity verification, with multiple opportunities for extension in response to evolving storage technologies and forensic requirements. Additionally, the activation function formalism is designed to be extended to further file systems. Defining instantiations for file systems such as APFS, Btrfs, and XFS, along with a corresponding empirical evaluation, constitutes a natural next step.

9. Conclusion

The growth of data volumes (Hargreaves et al., 2024) and the non-deterministic behavior of SSDs under TRIM/Deallocate operations (Bonetti et al., 2013; Neyaz et al., 2019) render traditional bitwise integrity verification increasingly impractical. To address these challenges, we introduced and formalized *active block hashing*, a method for evidential integrity verification that enables localization of modifications, improves robustness for SSDs through TRIM invariance, and, on partially filled media, reduces computational effort. This deterministic hashing scheme achieves these properties by parsing FS structures to identify and hash only those blocks that contain meaningful data. By organizing commitments to groups of active blocks in an MT, the method supports localization of modifications and partial integrity verification without recomputing a full digest. We described concrete instantiations for FAT, NTFS, and ext FSs and provided an [open-source implementation](#), `ahsum`.

We evaluated active block hashing with respect to technical correctness, TRIM invariance, and runtime performance. Runtime was compared against two variants of file hashing—FS driver-mediated and FS parser-mediated—as well as against traditional bit-by-bit hashing, using widely adopted open-source tools. The results show that, in linear and CMT modes, active block hashing is more efficient than bitwise hashing for media with fill levels up to approximately 91–97 %, depending on the file system. It also significantly outperforms FS driver-mediated file hashing on non-empty media and performs on par with FS parser-mediated file hashing, while protecting the integrity of a larger portion of metadata.

The systematic comparison of these integrity verification approaches and the experimental results indicate that active block hashing is a practical and effective complement to existing methods, offering a balanced trade-off between integrity protection, computational efficiency, and verification granularity.

CRedit authorship contribution statement

Jan Gruber: Conceptualization, Funding acquisition, Methodology, Investigation, Software, Writing - original draft, Writing - review & editing, Validation, Visualization.

Felix Freiling: Conceptualization, Funding acquisition, Methodology, Writing - review & editing.

Simson L. Garfinkel: Conceptualization, Methodology, Writing - review & editing.

Data availability

The prototype implementation, together with the scripts and evaluation pipeline used to produce the reported results, is available at <https://github.com/jgru/active-block-hashing> under the GNU General Public License v3.0.

A Runtime measurements

Tables 5–7 list the mean runtimes of all tools evaluated in Section 6 across fill levels 0–100 % and partition counts on two 128 GB SATA III SSDs with FAT32, ext4, and NTFS, respectively.

Table 5

Average runtime (in seconds) of tools hashing 128 GB SSDs (attached via SATA III) with a FAT32 FS across partitions and fill levels.

Tool	# P	Fill Level (%)										
		0	10	20	30	40	50	60	70	80	90	100
CMT ahsun ($\eta = 256, k = 2$)	1	0.264	31.273	62.291	93.483	124.299	155.412	186.295	217.493	248.681	279.678	312.319
	4	0.514	31.675	62.798	93.885	125.244	156.383	187.202	218.453	249.516	280.904	311.939
Lin. ahsun	1	0.275	31.000	61.521	92.333	122.869	153.489	184.183	215.066	245.971	276.576	307.142
	4	0.520	31.249	61.939	92.601	123.231	155.560	184.914	215.627	246.379	277.139	307.728
SPMT ahsun ($\eta = 256, k = 2$)	1	242.636	249.680	256.516	263.373	270.169	276.906	283.953	290.842	297.873	304.704	314.449
	4	242.768	249.974	256.949	263.843	270.664	277.697	284.320	292.092	298.604	305.574	312.296
fiwalk	1	0.037	33.331	66.491	99.618	132.849	166.235	199.568	232.829	266.236	299.533	335.935
	4	0.044	33.378	66.611	99.798	133.099	166.241	199.396	234.779	265.786	299.238	332.685
sha256deep	1	0.016	53.816	107.288	160.891	214.209	269.827	321.333	374.923	428.435	483.583	535.388
	4	0.014	53.928	107.319	161.027	214.563	268.126	323.059	375.223	428.754	482.280	535.737
sha256sum	1	281.379	282.710	281.890	282.886	282.485	282.881	283.102	283.641	284.600	284.549	284.724
	4	281.530	281.862	282.088	283.072	282.604	284.652	283.407	283.721	283.980	284.122	284.600

Table 6

Average runtime (in seconds) of tools hashing 128 GB SSDs (attached via SATA III) with an ext4 FS across partitions and fill levels.

Tool	# P	Fill Level (%)										
		0	10	20	30	40	50	60	70	80	90	100
CMT ahsun ($\eta = 256, k = 2$)	1	7.197	35.893	64.831	93.570	122.042	150.227	178.912	207.443	236.137	264.834	292.627
	4	7.780	36.282	65.038	93.563	122.152	151.967	179.327	208.097	236.458	265.244	293.058
Lin. ahsun	1	7.187	35.820	64.690	93.340	121.748	149.857	178.947	206.810	235.406	263.950	291.842
	4	7.742	36.162	64.773	93.333	121.863	151.478	178.742	207.344	235.885	264.565	292.363
SPMT ahsun ($\eta = 256, k = 2$)	1	243.650	249.067	253.511	258.607	263.468	268.283	273.496	278.481	283.886	288.793	292.743
	4	243.797	248.885	253.910	259.147	263.539	268.877	273.755	278.554	284.188	289.277	293.104
fiwalk	1	0.039	29.180	58.546	88.090	117.260	147.049	176.692	206.791	236.327	266.060	297.314
	4	0.054	29.326	58.687	87.943	118.384	146.708	176.007	205.454	235.132	264.890	296.695
sha256deep	1	0.013	52.577	104.864	157.827	209.436	263.920	315.274	367.941	420.447	470.800	525.945
	4	0.019	52.503	104.918	157.167	211.440	262.328	314.778	365.992	418.112	470.358	522.939
sha256sum	1	281.319	281.169	281.498	281.800	281.845	282.073	284.576	282.501	283.779	283.330	283.509
	4	281.646	281.493	281.734	282.072	282.178	285.022	282.229	282.706	283.002	283.289	284.957

Table 7

Average runtime (in seconds) of tools hashing 128 GB SSDs (attached via SATA III) with an NTFS FS across partitions and fill levels.

Tool	# P	Fill Level (%)										
		0	10	20	30	40	50	60	70	80	90	100
CMT ahsum ($\eta = 256, k = 2$)	1	0.815	30.042	59.249	88.542	117.746	146.942	176.252	207.035	234.853	263.202	292.446
	4	1.322	30.510	59.689	88.870	118.056	147.264	177.529	205.541	234.810	263.346	292.630
Lin. ahsum	1	0.823	29.919	59.011	88.435	117.243	146.413	175.511	206.313	233.956	262.373	291.264
	4	1.325	30.494	59.650	88.888	118.059	147.259	177.858	205.557	234.494	263.233	293.401
SPMT ahsum ($\eta = 256, k = 2$)	1	242.611	247.176	252.553	257.632	263.076	268.364	273.463	279.643	284.255	288.087	292.993
	4	242.547	247.221	252.452	257.640	262.848	267.878	272.980	278.114	283.310	287.952	292.792
fiwalk	1	0.040	29.162	58.277	87.397	116.586	145.983	175.181	204.217	233.600	263.297	293.447
	4	0.045	29.229	58.310	87.460	116.755	145.769	175.269	204.996	232.987	262.337	291.426
sha256deep	1	0.011	52.552	105.059	157.663	210.173	262.534	317.982	367.910	420.092	475.432	529.117
	4	0.014	53.337	105.277	158.129	211.340	264.339	316.590	369.306	420.477	478.780	529.824
sha256sum	1	281.346	281.678	281.913	285.468	282.571	282.855	283.130	284.771	283.670	283.911	283.768
	4	281.651	281.784	282.583	282.610	282.636	282.440	285.877	283.178	285.304	284.440	283.651

B Correctness (proof sketch)

We argue that the constructions of Sections 4.3 and 4.4 satisfy the four requirements of Section 4.2 under the assumption that the hash function H is collision-resistant and that the underlying FS is structurally intact and internally consistent (Section 7.3). The argument treats SPMT mode as the general case; CMT mode instead groups the compacted sequence of active DUs and contains no padding leaves—the same injectivity and domain-separation arguments apply—and linear mode degenerates to a single hash invocation. The core reduction follows the standard Merkle tree security argument (Merkle, 1989).

Positional binding

Goal: each active DU is bound to its logical position, so no reordering or relocation of active DUs can preserve the root hash. Each leaf serializes fixed-width tuples $(i \parallel d)$, i.e., logical block address and content. Fixed widths make the concatenation injective at both the tuple and group level: the number of tuples in $\tilde{g}_x$ is recoverable from the input length. Across groups, each tuple embeds its absolute address i at a fixed width, so a group's serialization uniquely determines the tuples it contains; since distinct groups draw from disjoint address ranges, no two distinct groups can produce the same serialization. Any leaf collision therefore reduces to a collision in H . Padding leaves $H(\gamma)$ carry no positional data; their positions are determined by the binding argument below. In linear mode, the same argument applies to the single concatenation using physical addresses, whose disjointness holds across partition boundaries.

Structural soundness

Goal: the aggregation structure is unambiguous, so distinct tree structures cannot yield the same root hash. Pairwise-distinct domain-separation tags α (leaves, including linear-mode roots), β (internal nodes), γ (padding), and δ (partition-root aggregation) partition all hash inputs into disjoint categories, so no cross-category collision can occur except via a collision in H (Sections 4.3 and 4.4). The tree is a deterministic k -ary tree whose shape is uniquely determined by the number of leaves; children have fixed hash length λ , so the child count at every internal node is recoverable from the input length. Any structural rearrangement of active DUs therefore changes at least one hash input and, under collision resistance of H , the root. In SPMT mode, all padding leaves equal $H(\gamma)$; permuting them yields an identical tree, and whether a position holds padding or data is committed by the binding argument below.

Binding to logical acquisition state

Goal: the single root hash binds the complete logical acquisition state—the active DUs, their ordering and content, and their activation classification. Every supported activation function A , classifies all FS metadata DUs (superblocks, allocation bitmaps, etc.) as active—either because they are fixed regions derived from the boot sector or superblock, or because an internally consistent instance marks its own metadata as allocated (Section 3.3). Because these DUs are always committed in the leaf hashes, and the activation status of all remaining DUs is a deterministic function of the committed metadata, the commitment is well-founded with no circular dependency. Any reclassification of a DU requires altering committed metadata and thus changes the root hash. Conversely, because the activation classification depends only on committed metadata and never on the content of inactive DUs, modifications confined to inactive DUs cannot affect any metadata DU and therefore leave the logical acquisition state—and hence the root hash—unchanged by design. Combined with positional binding and structural soundness, the root hash binds the complete ordered sequence of active DUs and their contents; in SPMT mode, padding-leaf positions are additionally fixed by κ and η .

Deterministic verifiability

Goal: any verifier can, from the serialized state and public parameters alone, deterministically reconstruct the identical tree and root hash. The activation function A_τ (or A'_τ), file-system type τ , block size b , grouping parameter η , arity k , hash function H , domain-separation tags, and the mode of operation are public parameters (κ is derived from the image size, τ , and b). Since A_τ is deterministic and public, any verifier can, under the intactness assumption above, reconstruct the identical tree and root hash from the storage image.

In the tree-based modes, the per-partition roots are aggregated into a global root via the same k -ary internal-node recursion over the ordered sequence of λ -bit roots, but under the distinct aggregation tag δ (Section 4.3); a collision at any aggregation node is therefore directly a collision in H , since the concatenation of fixed-width children is injective. Crucially, because $\delta \neq \beta$, an aggregation node can never coincide with a within-partition internal node. The aggregation layer is itself a δ -tagged k -ary tree whose shape is determined by the number of partitions, so distinct partition counts also yield distinct roots. Partitions whose roots coincide have, under collision resistance, identical leaf serializations, so their relative order is immaterial. Taken together, any tree-mode root collision implies, by the standard MT reduction, a collision at some node; domain separation ensures this across all input categories. For linear mode, the single device-wide α -tagged serialization is injective, so a root collision is directly a collision in H ; because linear mode emits no β -, γ -, or δ -tagged nodes, no intra-mode structural ambiguity can arise. Since the mode of operation is a public parameter, cross-mode collisions are excluded. Under collision resistance of H , the root hash thus constitutes a binding commitment to the logical acquisition state, for a fixed setting of the public parameters (τ , b , κ , η , k , mode, tags): any modification to the content of active DUs, their ordering, or their activation status produces a distinct root hash except via a collision in H . The reduction is constructive—any two distinct states sharing a root yield an explicit H -collision by descending to the first differing node—so finding a root-preserving modification is as hard as finding a collision in H .

References

- Altheide, C., Carvey, H., 2011. Digital Forensics with Open Source Tools. Elsevier.
- Bell, G.B., Boddington, R., 2010. Solid state drives: the beginning of the end for current practice in digital forensic recovery? Journal of Digital Forensics, Security and Law 5, 1–20. <https://doi.org/10.15394/jdfl.2010.1078>.
- Bonetti, G., Viglione, M., Frossi, A., Maggi, F., Zanero, S., 2013. A comprehensive black-box methodology for testing the forensic characteristics of solid-state drives. In: Payne Jr., C.N. (Ed.), Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9–13, 2013, ACM, pp. 269–278. <https://doi.org/10.1145/2523649.2523660>.
- Bray, T., 2017. The JavaScript object notation (JSON) data interchange format. RFC 8259. <https://doi.org/10.17487/RFC8259>.
- Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Educational, Boston, MA.
- Garfinkel, S.L., 2009. Automating disk forensic processing with SleuthKit, XML and Python. In: Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering, SADFE 2009, Berkeley, California, USA, May 21, 2009, IEEE Computer Society, pp. 73–84. <https://doi.org/10.1109/SADFE.2009.12>.
- Garfinkel, S.L., 2012. Digital forensics XML and the DFXML toolkit. Digit. Invest. 8, 161–174. <https://doi.org/10.1016/J.DIIN.2011.11.002>.
- Grier, J., Richard III, G.G., 2015. Rapid forensic imaging of large disks with sifting collectors. Digit. Invest. 14 (Suppl. 1), S34–S44. <https://doi.org/10.1016/J.DIIN.2015.05.006>.
- Gruber, J., Humml, M., 2023. A formal treatment of expressiveness and relevance of digital evidence. Digital Threats: Research and Practice 4 (3), 33:1–33:16. <https://doi.org/10.1145/3608485>.
- Hadi, H.J., Musthaq, N., Khan, I.U., 2021. SSD forensic: evidence generation and forensic research on solid state drives using TRIM analysis. In: 2021 International Conference on Cyber Warfare and Security (ICCCWS). IEEE, pp. 51–56. <https://doi.org/10.1109/iccws53234.2021.9702989>.
- Han, J., Han, M.L., Lee, S., Park, J., 2024. Eco-Bag: an elastic container based on Merkle tree as a universal digital evidence bag. Forensic Sci. Int. Digit. Invest. 49, 301725. <https://doi.org/10.1016/J.FSIDI.2024.301725>.
- Hargreaves, C., Breitering, F., Dowthwaite, L., Webb, H., Scanlon, M., 2024. DFPulse: the 2024 digital forensic practitioner survey. Forensic Sci. Int. Digit. Invest. 51, 301844. <https://doi.org/10.1016/J.FSIDI.2024.301844>.
- King, C., Vidas, T., 2011. Empirical analysis of solid state disk data retention when used with contemporary operating systems. Digit. Invest. 8, S111–S117. <https://doi.org/10.1016/J.DIIN.2011.05.013>.
- Menezes, A.J., van Oorschot, P.C., Vanstone, S.A., 1996. Handbook of applied cryptography. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL.
- Merkle, R.C., 1979. Secrecy, Authentication and Public Key Systems. Ph.D. thesis. Stanford University.
- Merkle, R.C., 1989. A certified digital signature. In: Brassard, G. (Ed.), Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20–24, 1989, Proceedings. Springer, pp. 218–238. https://doi.org/10.1007/0-387-34805-0_21.
- Mitchell, I., Anandaraja, T., Hara, S., Hadzhiniov, G., Neilson, D., 2016. Deconstruct and Preserve (DaP): a method for the preservation of digital evidence on solid state drives (SSD). In: Jahankhani, H., Carlile, A., Emm, D., Hosseinian-Far, A., Brown, G., Sexton, G., Jamal, A. (Eds.), Global Security, Safety and Sustainability – the Security Challenges of the Connected World. Springer International Publishing, Cham, pp. 3–11. https://doi.org/10.1007/978-3-319-51064-4_1.
- Mitchell, I., Ferriera, J., Anandaraja, T., Hara, S., 2018. DaPv: Deconstruct and Preserve for All: a procedure for the preservation of digital evidence on solid state drives and traditional storage media. In: Jahankhani, H. (Ed.), Cyber Criminology. Springer International Publishing, Cham, pp. 275–287. https://doi.org/10.1007/978-3-319-97181-0_13.
- Neyaz, A., Zhou, B., Karpoor, N., 2019. Comparative study of wear-leveling in solid-state drive with NTFS file system. In: Baru, C.K., Huan, J., Khan, L., Hu, X., Ak, R., Tian, Y., Barga, R.S., Zaniolo, C., Lee, K., Ye, Y.F. (Eds.), 2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9–12, 2019. IEEE, pp. 4294–4298. <https://doi.org/10.1109/BIGDATA47090.2019.9006067>.
- Nisbet, A., Jacob, R., 2019. TRIM, wear levelling and garbage collection on solid state drives: a prediction model for forensic investigators. In: 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, August 5–8, 2019. IEEE, pp. 419–426. <https://doi.org/10.1109/TRUSTCOM/BIGDATASE.2019.00063>.
- Nisbet, A., Lawrence, S., Ruff, M., 2013. A forensic analysis and comparison of solid state drive data retention with TRIM enabled file systems. In: 11th Australian Digital Forensics Conference, Security Research Institute (SRI). Edith Cowan University, Western Australia, pp. 103–111. <https://doi.org/10.4225/75/57B3D766FB873>.
- Roussev, V., Quates, C., Martell, R., 2013. Real-time digital forensics and triage. Digit. Invest. 10, 158–167. <https://doi.org/10.1016/j.diin.2013.02.001>.
- Stüttgen, J., Dewald, A., Freiling, F.C., 2013. Selective imaging revisited. In: Morgenstern, H., Ehlert, R., Freiling, F.C., Frings, S., Göbel, O., Günther, D., Kiltz, S., Nedon, J., Schadt, D. (Eds.), Seventh International Conference on IT Security Incident Management and IT Forensics, IMF 2013, Nuremberg, Germany, March 12–14, 2013. IEEE Computer Society, pp. 45–58. <https://doi.org/10.1109/IMF.2013.16>.
- Turner, P., 2005. Unification of digital evidence from disparate sources (Digital Evidence Bags). Digit. Invest. 2, 223–228. <https://doi.org/10.1016/j.diin.2005.07.001>.