Integrity Verification of Digital File Collections Using Merkle Trees

Carlen Asadel Axelle - 18223017
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: carlenasadel@gmail.com , 18223017@std.stei.itb.ac.id

***Abstract*—Digital file collections are vulnerable to unnoticed modification, deletion, and insertion, especially when files are stored locally, archived, exchanged between users, or used as evidence in security-related workflows. This paper presents a Merkle Tree-based integrity verification application for digital file collections. The proposed system computes SHA-256 hashes for individual files, arranges them as leaves in a binary Merkle Tree, and produces a Merkle Root as a compact commitment to the entire folder state. A baseline manifest is stored as a text file containing the Merkle Root and per-file hash records. During verification, the application recomputes the current folder state and compares it with the baseline to detect modified, deleted, newly added, and size-changed files. The implementation is built using Python and Tkinter, with no external dependency beyond the Python standard library. The contribution of this work is the design and implementation of a GUI-based Merkle Tree verifier for file collections, supported by an experimental plan to evaluate correctness in detecting file changes and structural folder changes.**

Keywords—merkle tree; SHA-256; hash; file integrity; python

I. INTRODUCTION (*HEADING 1*)

Digital file collections are frequently used in academic work, software projects, digital archives, forensic evidence handling, and administrative document storage. In these contexts, the integrity of a collection is not limited to verifying one file. A practical system must also verify whether the whole folder remains unchanged after a baseline has been created. A single modified report, a deleted evidence file, or an inserted unauthorized file may change the meaning and reliability of the entire collection.

A common approach to file integrity checking is to compute a cryptographic hash for each file. However, when a folder contains many files, managing a long list of independent hashes becomes less compact and less convenient. A Merkle Tree solves this problem by organizing file hashes into a tree structure. The leaves represent individual file hashes, while internal nodes are computed from their children. The root of the tree, called the Merkle Root, becomes a compact representation of the complete collection state. If one file changes, the corresponding leaf hash changes, causing the parent hashes and eventually the Merkle Root to change.

This paper proposes and implements a GUI-based application titled Merkle Tree File Integrity Verifier. The application is designed to select a folder, compute SHA-256 hashes for all files, build a Merkle Tree, save a baseline manifest, import an existing manifest, generate Merkle Proof for a selected file, verify a proof against the Merkle Root, simulate file modification, and perform full folder verification. The implementation is intended for a local integrity verification scenario where the baseline manifest is trusted and later used to detect changes in the file collection.

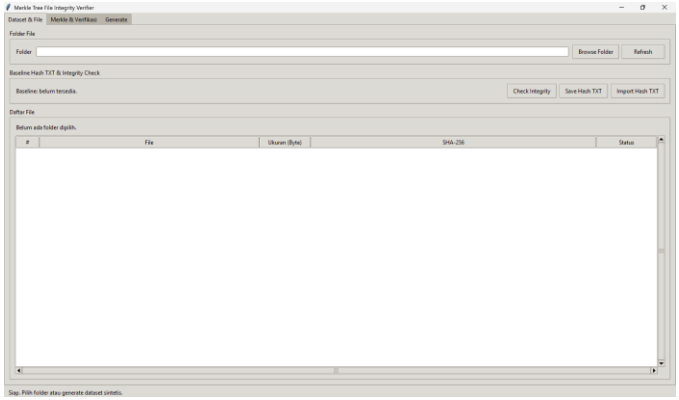


Fig. 1. Screenshot of the main GUI showing the “Dataset & File”, “Merkle & Verification”, and “Generate” tabs.

II. THEORETICAL BASIS

A. Cryptographic Hash Function

A cryptographic hash function maps an input of arbitrary length into a fixed-length digest. In the context of file integrity, a hash digest acts as a compact representation of file content. If the file changes, even by a small amount, a secure hash function is expected to produce a different digest. This property makes hash functions useful for detecting unauthorized or accidental changes in digital data [3].

This project uses SHA-256, which is part of the SHA-2 family specified in the Secure Hash Standard. SHA-256 produces a 256-bit digest. The digest is represented as a hexadecimal string in the implementation. In this application, SHA-256 is used at two levels: first, to hash the content of each file, and second, to hash the concatenation of two child hashes when constructing internal Merkle Tree nodes.

The integrity security of the system depends on the collision resistance and second-preimage resistance of the hash function. Collision resistance means that it should be computationally difficult to find two different inputs with the same digest. Second-preimage resistance means that, given one valid input, it should be difficult to find another different input with the same digest. For this application, second-preimage resistance is especially important because an attacker should not be able to replace a file with different content while keeping the same hash value [4].

B. Merkle Tree

A Merkle Tree is a tree-based authenticated data structure. The leaves contain hashes of data blocks, while each internal node is computed from the hash values of its children. The root hash commits to the entire set of leaves. If any leaf changes, the change propagates upward and results in a different root. Merkle's tree authentication concept allows the correctness of a data item to be verified using only the item, the root, and a limited number of intermediate hash values rather than requiring the full dataset [1].

In a binary Merkle Tree, each parent node is computed from two child nodes. If the number of leaves is not even, implementations may apply a specific policy to handle the unpaired leaf. In this project, the chosen policy is to duplicate the last leaf when a level has an odd number of nodes. Therefore, every internal node is always computed from a pair of hashes.

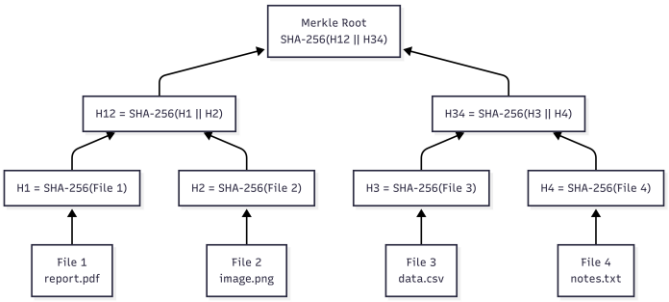


Fig. 2. Diagram of a binary Merkle Tree

The Merkle Root acts as a compact commitment to a folder state. A folder state in this paper is defined by the ordered list of relative file paths, file sizes, and SHA-256 file hashes recorded during baseline generation. If the folder is later verified, the application recomputes the file hashes and the Merkle Root. If the new root differs from the baseline root, the folder integrity is considered invalid.

The Merkle Root itself does not reveal which file changed. Therefore, this project also stores a manifest containing per-file hash records. The manifest enables the verifier to compare old and current file records and classify changes into categories such as MODIFIED, SIZE_CHANGED, NEW, and DELETED.

C. Merkle Root

A Merkle Proof, also known as an inclusion proof or authentication path, is a list of sibling hashes needed to recompute the root from a selected leaf. Instead of recomputing the entire tree, a verifier can hash the selected file, combine it with each sibling hash in the proof, and compare the resulting root against the trusted Merkle Root. RFC 9162 describes Merkle inclusion proof as the shortest list of additional nodes required to compute the Merkle Tree Hash for a specific tree [2].

In this project, a proof step contains the level number, sibling position, sibling index, and sibling hash. The sibling position is important because hash concatenation is order-sensitive. If the sibling is on the left, the application computes:

```
SHA256(sibling_hash || current_hash)
```

If the sibling is on the right, the application computes:

```
SHA256(current_hash || sibling_hash)
```

The proof is valid if the final computed root equals the expected Merkle Root.

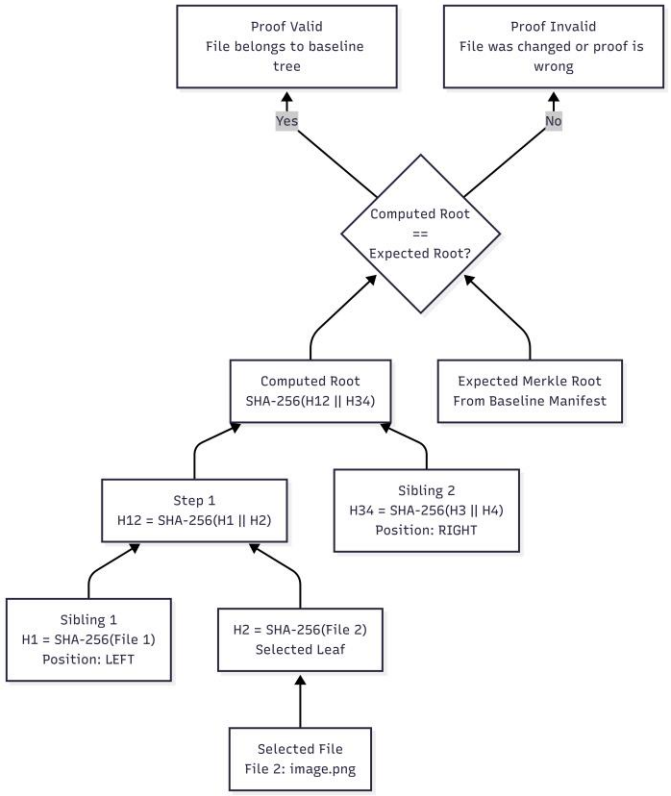


Fig. 3. Ilustration of Merkle Proof verification for one selected file.

III. Proposed Method

A. System Overview

The proposed system verifies the integrity of a digital file collection using a baseline-and-verification workflow. The baseline is generated when the folder is assumed to be in a valid state. The application scans the folder, hashes every file, builds the Merkle Tree, displays the Merkle Root, and saves a manifest file. At a later time, the user can import the manifest and verify whether the folder still matches the baseline.

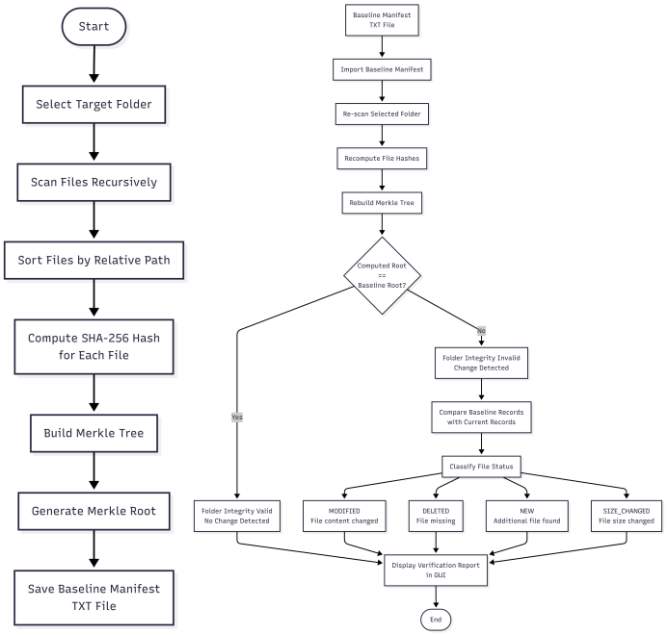


Fig. 4. System workflow diagram

B. Folder Scanning and Ordering

The folder is scanned recursively so that files inside subfolders are also included. Each file is represented by its relative path, absolute path, size in bytes, and SHA-256 hash. Relative paths are used because they make the manifest more portable than absolute paths. The implementation sorts files by their relative path in lowercase form to ensure deterministic ordering.

Deterministic ordering is important because the Merkle Root depends not only on file content but also on the order of leaves. If the same files are processed in different orders, the resulting root may be different even when file contents are identical. Therefore, the proposed method uses a stable ordering rule before building the leaf list.

TABLE I.
FILE RECORD FIELDS USED IN THE BASELINE MANIFEST

Field	Description
index	Deterministic file index after sorting
relative_path	File path relative to selected folder
absolute_path	Full local file path used during current execution
size_bytes	File size in bytes
sha256	SHA-256 digest of file content
status	Integrity status during comparison

C. File Hashing

Each file is opened in binary mode and read in chunks. This avoids loading the entire file into memory at once, which is more practical for larger files. The default chunk size is 64 KB. For each chunk, the SHA-256 digest object is updated until the file ends. The final hexadecimal digest is stored in the file record.

The file hash computation is expressed as:

```
H_file = SHA256(file_bytes)
```

where `H_file` is the leaf hash used in the Merkle Tree.

D. Merkle Tree Construction

After all file hashes are computed, the application builds the Merkle Tree. The leaf level is the ordered list of SHA-256 file hashes. Each parent is computed by concatenating the byte representation of two child hashes and hashing the result with SHA-256:

```
H_parent = SHA256(bytes(left_hash) || bytes(right_hash))
```

If a level contains an odd number of nodes, the last hash is duplicated:

```
H_parent = SHA256(bytes(last_hash) || bytes(last_hash))
```

This policy ensures that each parent node always has two inputs.

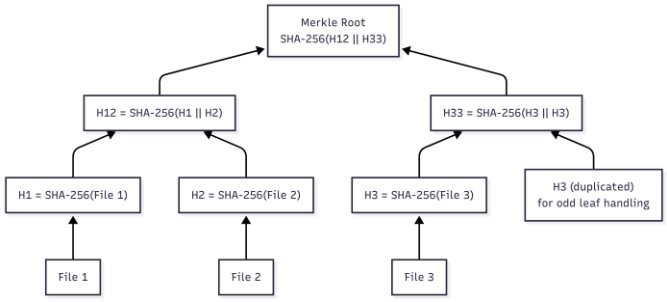


Fig. 5. Odd leaf duplication policy
F

The final single hash is stored as the Merkle Root. If the folder contains no file, the application does not create a Merkle Tree and returns an error because there is no leaf to commit.

E. Merkle Tree Construction

The baseline manifest is saved as a text file. The manifest contains metadata and the list of file records. The metadata includes the manifest version, hash algorithm, tree pair hash rule, odd leaf policy, Merkle Root, creation time, and file count.

TABLE II.
BASELINE MANIFEST METADATA FIELDS

Metadata Field	Value / Description
Manifest header	Deterministic file index after sorting

Metadata Field	Value / Description
Algorithm	File path relative to selected folder
Tree pair hash	sha256(bytes.fromhex(left)+bytes.fromhex(right))
Odd leaf policy	duplicate_last
Merkle Root	Root hash of the baseline folder
Created at	Timestamp when manifest is created
File count	Number of files in baseline

The manifest is important because the Merkle Root alone can detect that a change occurred, but it cannot classify which file was modified, deleted, or added. By storing per-file records, the application can compare the baseline and current folder state in more detail.

F. Integrity Verification

The verification phase compares the current folder state with the baseline manifest. The application imports the manifest, scans the selected folder again, recomputes file hashes, rebuilds the Merkle Tree, and compares the newly computed root with the expected root from the manifest.

If the roots are equal, the folder is considered valid. If the roots are different, the folder is considered invalid. The application then compares baseline records and current records using relative paths as keys. The comparison classifies each file into one of the following statuses:

TABLE III.
FILE CHANGE CLASSIFICATION USED BY THE VERIFIER

Status	Condition
UNCHANGED	File exists in both baseline and current folder, and the hash is identical
MODIFIED	File exists in both states, size is identical, but hash is different
SIZE_CHANGED	File exists in both states, but size and hash differ
NEW	File exists in current folder but not in baseline
DELETED	File exists in baseline but not in current folder

G. Merkle Proof Verification

The application can also generate a Merkle Proof for a selected file. The proof contains sibling hashes along the path from the selected leaf to the root. During proof verification, the selected file is hashed again and combined with the proof steps. If the recomputed root equals the expected root, the selected file is proven to be part of the baseline tree.

This feature demonstrates the efficient verification property of Merkle Trees. Instead of requiring all file hashes to verify one file, the verifier only needs the selected file hash, its authentication path, and the trusted Merkle Root [1], [2].

Fig. 6. Merkle Proof text area showing level, sibling position, sibling index, and sibling hash Implementation

H. Development Environment

The application is implemented using Python and Tkinter. Python is used for file handling, hashing, data structure implementation, manifest processing, and unit testing. Tkinter is used to provide a desktop GUI so that users can interact with the verifier without executing manual commands. The program does not require external dependencies beyond the Python standard library.

TABLE IV.
IMPLEMENTATION ENVIRONMENT

Component	Description
Programming language	Python
GUI library	Tkinter
Hash library	Hashlib
Hash algorithm	SHA-256
Manifest format	TXT with metadata and tab-separated file records
Testing framework	unittest
External dependency	None

I. Project Structure

The project is divided into several modules to separate GUI, cryptographic logic, manifest handling, dataset generation, and testing.

TABLE V.
PROJECT STRUCTURE

File / Folder	Responsibility
app.py	Application entry point
gui/main_window.py	Tkinter GUI and user interaction flow
core/hash_utils.py	File scanning and SHA-256 hashing
core/merkle_tree.py	Merkle Tree construction and Merkle Proof logic
core/verifier.py	Full folder verification and change comparison
core/manifest.py	Manifest creation, saving, loading, and comparison
dataset/generator.py	Synthetic dataset generation and file modification simulation
tests/test_core.py	Unit tests for hashing, Merkle Tree, proof, and verification

File / Folder	Responsibility
tests/test_generator.py	Unit tests for dataset generation

J. Graphical User Interface

The GUI consists of three main tabs: **Dataset & File**, **Merkle & Verification**, and **Generate**.

The **Dataset & File** tab allows the user to select a folder, refresh the file list, save a baseline hash manifest, import an existing manifest, and check folder integrity. It also displays file records in a table with columns for index, relative path, size, SHA-256 hash, and status.

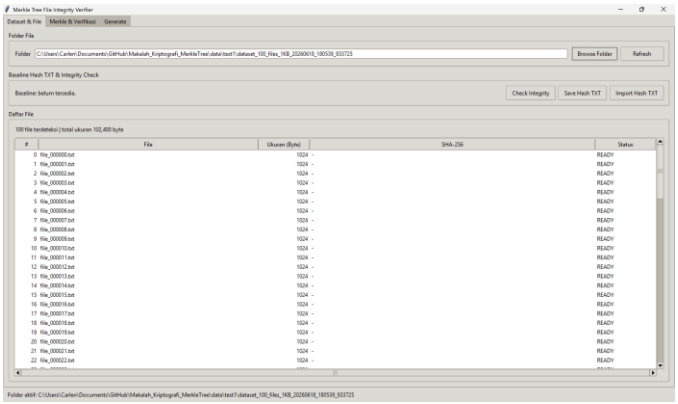


Fig. 7. Dataset & File tab after selecting a folder

The **Merkle & Verification** tab allows the user to build a Merkle Tree, view the Merkle Root, run full verification, generate a Merkle Proof for a selected file, verify the proof, and simulate file modification.

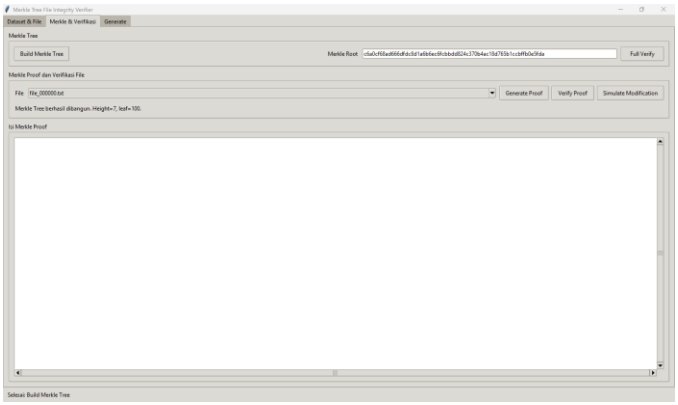


Fig. 8. Merkle & Verification tab after building a Merkle Tree

The **Generate** tab allows the user to create synthetic datasets. The default configuration supports combinations of file counts such as 10, 100, 500, and 1000 files, and file sizes such as 1 KB, 10 KB, 100 KB, and 1024 KB per file.

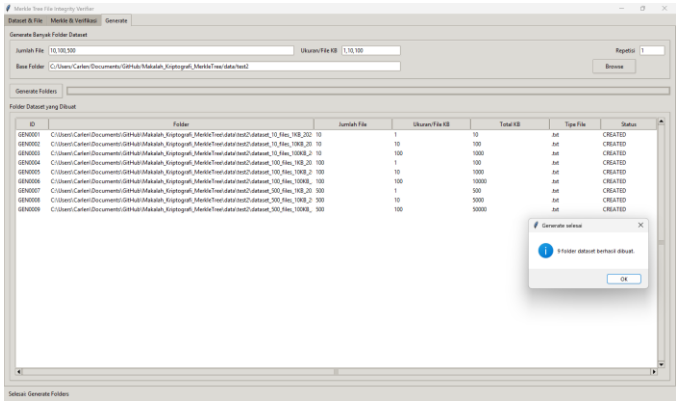


Fig. 9. Generate tab with synthetic dataset parameters

K. Core Algorithm

The implemented algorithm can be summarized as follows:

1. User selects a folder.
2. Program scans all files recursively.
3. Files are sorted by relative path.
4. Program computes SHA-256 hash for every file.
5. Hashes are inserted as Merkle Tree leaves.
6. Program computes parent hashes until a single Merkle Root remains.
7. Program creates a manifest containing the root and file hash records.
8. During verification, program recomputes the current root.
9. Program compares the current root with the baseline root.
10. If different, program classifies file-level changes.

Pscode. 1. Baseline generation algorithm

```
Input: folder_path
Output: manifest

files = scan_files_recursively(folder_path)
files = sort_by_relative_path(files)

records = []
for each file in files:
    hash_value = SHA256(file.content)
    records.append(index, relative_path,
size_bytes, hash_value)

tree = build_merkle_tree(records.hash_values)
root = tree.root

manifest = create_manifest(
    algorithm = "SHA-256",
    merkle_root = root,
    file_count = len(records),
```

```
        records = records
    )

    save_manifest_txt(manifest)
return manifest
```

Pscode. 2. Integrity verification algorithm

```
    Input:folder_path, baseline_manifest
    Output: verification_result

    current_records = hash_files(folder_path)
    current_tree =
build_merkle_tree(current_records.hash_values)
    computed_root = current_tree.root

    if computed_root ==
baseline_manifest.merkle_root:
        result = VALID
    else:
        result = INVALID

    changes = compare_records(
        baseline_manifest.records,
        current_records
    )

return result, computed_root, changes
```

Pscode. 3. Merkle Proof verification algorithm

```
    Input: file_path, proof, expected_root
    Output: valid or invalid

    current_hash = SHA256(file_path.content)

    for each step in proof:
        if step.sibling_position == "left":
            current_hash =
SHA256(step.sibling_hash || current_hash)
        else if step.sibling_position == "right":
            current_hash = SHA256(current_hash ||
step.sibling_hash)

return current_hash == expected_root
```

L. *Dataset Generator*

The dataset generator creates synthetic .txt files containing random alphanumeric characters. The user can specify the number of files, file size in KB, repetition count, and target base folder. Each dataset folder is named using the number of files, file size, and timestamp. This feature supports repeatable experimentation because different dataset sizes can be generated from the GUI.

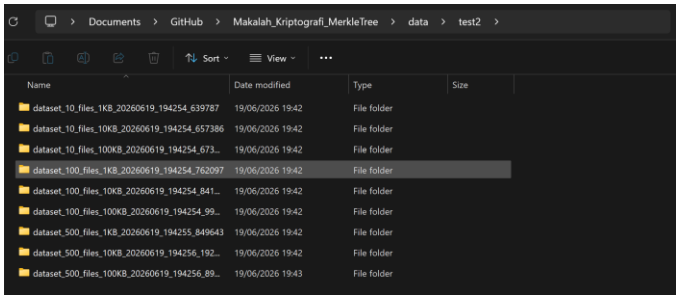


Fig. 10. Generated synthetic dataset folders

IV. Experiment Design

A. *Experiment Objective*

The experiments are designed to evaluate whether the implemented Merkle Tree verifier can correctly detect integrity changes in digital file collections. Since the main focus of this paper is correctness rather than performance optimization, the experiment emphasizes change detection scenarios involving modified files, deleted files, and newly added files.

The experiments answer the following questions:

1. Can the system detect a file whose content has been modified?
2. Can the system detect a file that has been deleted after baseline generation?
3. Can the system detect a new file added after baseline generation?
4. Can the system distinguish different change categories using the manifest?
5. Does the Merkle Root change whenever the folder content changes?

B. *Dataset Configuration*

The experiments are designed to evaluate whether the implemented Merkle Tree verifier can correctly detect integrity changes in digital file collections. Since the main focus of this paper is correctness rather than performance optimization, the experiment emphasizes change detection scenarios involving modified files, deleted files, and newly added files.

Synthetic datasets are generated using the built-in generator. Each file contains random alphanumeric text. The experiments may use several combinations of file counts and file sizes to observe whether the verification result remains correct across different folder sizes.

TABLE VI.
PLANNED SYNTHETIC DATASET CONFIGURATION

Dataset ID	Number of Files	File Size per File
D1	10	1 KB
D2	100	1 KB
D3	100	10 KB
D4	500	1 KB
D5	1000	1 KB

C. Test Scenarios

The experiment uses four main scenarios. In each scenario, the baseline manifest is created first. Then, the file collection is changed according to the scenario. Finally, the application performs full verification and records the result.

TABLE VII.
INTEGRITY VERIFICATION TEST SCENARIO

Scenario ID	Scenario
S1	No file is changed after baseline
S2	One file is modified
S3	One file is deleted
S4	One new file is added

D. Test Scenarios

The experimental procedure is as follows:

1. Generate a synthetic dataset.
2. Select the generated folder in the application.
3. Build the Merkle Tree.
4. Save the baseline manifest.
5. Record the baseline Merkle Root.
6. Apply a controlled change according to the scenario.
7. Import the baseline manifest.
8. Run Check Integrity or Full Verify.
9. Record the computed Merkle Root and detected file statuses.
10. Repeat for each dataset configuration.

E. Test Result

TABLE VIII. EXPERIMENTAL RESULTS FOR UNCHANGED FILES

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D1	-	8e44590b78104759	8e44590b78104759	UNCHANGED
D2	-	313dc0a814937ca3	313dc0a814937ca3	UNCHANGED
D3	-	257ad9d42887ba66	257ad9d42887ba66	UNCHANGED
D4	-	56dbd7417f20f07d	56dbd7417f20f07d	UNCHANGED
D5	-	830eff64d514fa8f	830eff64d514fa8f	UNCHANGED

TABLE IX. EXPERIMENTAL RESULTS FOR MODIFIED FILES

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D1	file_000005.txt	8e44590b78104759	ee350c73044c4d60	MODIFIED
D2	file_000050.txt	313dc0a814937ca3	c342da2044ff8b7d	MODIFIED
D3	file_000050.txt	257ad9d42887ba66	5d722bbdcc1ee59c	MODIFIED
D4	file_000250.txt	56dbd7417f20f07d	f5f309ad1a287fd6	MODIFIED
D5	file_000500.txt	830eff64d514fa8f	492d0271ae783e54	MODIFIED

TABLE X. EXPERIMENTAL RESULTS FOR APPENDED FILES

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D1	file_000005.txt	8e44590b78104759	fedf894d2a76b008	SIZE_CHANGED
D2	file_000050.txt	313dc0a814937ca3	c5e8c7a5bdb13af9	SIZE_CHANGED
D3	file_000050.txt	257ad9d42887ba66	c54c455ab1198f67	SIZE_CHANGED
D4	file_000250.txt	56dbd7417f20f07d	25919376cad76061	SIZE_CHANGED
D5	file_000500.txt	830eff64d514fa8f	3dcbe53afbfc18b4	SIZE_CHANGED

TABLE XI. EXPERIMENTAL RESULTS FOR DELETED FILES

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D1	file_000005.txt	8e44590b78104759	0480fee0ccccc9db	DELETED

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D2	file_000050.txt	313dc0a814937ca3	ecaff10c6b92769f	DELETED
D3	file_000050.txt	257ad9d42887ba66	17717e22d0e23d4d	DELETED
D4	file_000250.txt	56dbd7417f20f07d	55450f7f41260eb8	DELETED
D5	file_000500.txt	830eff64d514fa8f	81de85206d7dc73e	DELETED

TABLE XII. EXPERIMENTAL RESULTS FOR NEWLY ADDED FILES

Dataset ID	Modified File	Baseline Root Prefix	Computed Root Prefix	Status
D1	file_new_s4.txt	8e44590b78104759	d2c92506572a60d2	NEW
D2	file_new_s4.txt	313dc0a814937ca3	b2a9bc4ffe14e39a	NEW
D3	file_new_s4.txt	257ad9d42887ba66	71c35da36d0b2de1	NEW
D4	file_new_s4.txt	56dbd7417f20f07d	712a172ec1406dcf	NEW
D5	file_new_s4.txt	830eff64d514fa8f	d124c952f237f040	NEW

V. ANALYSIS

A. *Integrity Detection Analysis*

The proposed system should detect file modification because any change in file content changes the SHA-256 digest of that file. Since the file hash is used as a Merkle Tree leaf, the changed leaf produces different parent hashes until the Merkle Root also changes. Therefore, a modified file is expected to make the computed root different from the baseline root.

File deletion is also expected to be detected. When a file is deleted, its relative path and hash record no longer appear in the current folder scan. This changes the leaf list and the number of leaves. Consequently, the recomputed Merkle Root differs from the baseline root. The manifest comparison also identifies the missing relative path and classifies it as DELETED.

A newly added file should produce a similar invalid root result. The added file introduces a new leaf into the Merkle Tree. Because the leaf list changes, the tree structure and Merkle Root also change. The record comparison identifies the new relative path as NEW.

The manifest-based comparison is necessary because the Merkle Root only indicates whether the collection is valid or invalid. It does not directly explain the cause of invalidity. By storing the baseline file records, the application can provide more useful information to the user, including which file was changed and what type of change occurred.

B. *Merkle Proof Analysis*

The Merkle Proof feature demonstrates selective verification. For a selected file, the application generates the list of sibling hashes needed to recompute the root. If the selected file is unchanged, the recomputed root equals the baseline root. If the selected file has been modified, its new SHA-256 hash causes the recomputed root to differ from the expected root.

This behavior is consistent with the authentication path concept described in Merkle Tree literature. Instead of verifying all files, a verifier only needs the selected file, the proof path, and the trusted root [2]. This is useful when a system needs to prove membership of one file in a known baseline without transmitting the entire file collection.

VI. CONCLUSION

This paper presents a Merkle Tree-based application for verifying the integrity of digital file collections. The system computes SHA-256 hashes for files, builds a binary Merkle Tree, generates a Merkle Root, and saves a baseline manifest containing file records and metadata. During verification, the current folder is rehashed and compared against the baseline to detect file modification, deletion, and addition.

The implementation demonstrates how Merkle Trees can be applied beyond blockchain or certificate transparency use cases. In this project, the Merkle Root is used as a compact commitment to a local folder state, while the manifest enables detailed file-level change classification. The GUI-based implementation also makes the workflow easier to use and understand.

Although the system provides effective integrity verification under a trusted-baseline assumption, it does not yet authenticate the manifest itself. Therefore, future development should add digital signatures or MAC protection for the manifest. Additional improvements may include real-time monitoring, performance benchmarking, and support for multiple hash algorithms.

VIDEO LINK AT YOUTUBE

https://youtu.be/StIfVjHHCbs

ACKNOWLEDGMENT

I would like to express sincere gratitude to **Dr. Ir. Rinaldi Munir, M.T.**, as the lecturer on the Cryptography class for his guidance, materials, and insights throughout the semester. I also thank all classmates in the Cryptography course for the discussions, feedback, and shared learning experiences that helped enrich the understanding of cryptographic concepts. Special appreciation is also given to my former groupmates in the course project for their collaboration, technical discussions, and support, which contributed to the development of ideas related to this paper.

REFERENCES

[1] [1] R. C. Merkle, “Protocols for public key cryptosystems,” in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1980, pp. 122–134.

[2] [2] B. Laurie, A. Langley, E. Kasper, E. Messeri, and R. Stradling, “Certificate Transparency Version 2.0,” RFC 9162, Dec. 2021, doi: 10.17487/RFC9162.

[3] [3] National Institute of Standards and Technology, “Secure Hash Standard (SHS),” FIPS PUB 180-4, Aug. 2015, doi: 10.6028/NIST.FIPS.180-4.

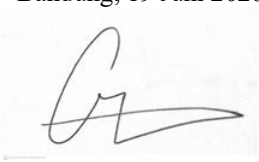
[4] [4] Q. H. Dang, “Recommendation for Applications Using Approved Hash Algorithms,” NIST Special Publication 800-107 Revision 1, Aug. 2012, doi: 10.6028/NIST.SP.800-107r1.

[5] [5] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, 1996.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026



Carlen Asadel Axelle
18223017