

Analisis Efisiensi Merkle Tree dalam Verifikasi Integritas Chunk File pada Simulasi Sistem Penyimpanan Terdistribusi

Justin William - 13525042

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: justinwilliam980@gmail.com , 13525042@std.stei.itb.ac.id

Abstract—Perkembangan sistem penyimpanan terdistribusi membutuhkan adanya mekanisme verifikasi integritas data, terutama ketika data disimpan secara terdistribusi ke banyak simpul. Metode verifikasi integritas data secara konvensional umumnya memerlukan pemeriksaan *file* sehingga biaya dan waktu juga meningkat seiring bertambahnya jumlah *chunk*. Oleh karena itu, salah satu metode yang lebih efisien dan dapat menangani permasalahan tersebut adalah dengan menggunakan struktur data *Merkle Tree*, yaitu sebuah struktur data berbasis *Binary Tree* yang menggunakan fungsi *hash* untuk merepresentasikan integritas dari sekumpulan data menggunakan *root hash*. Makalah ini menganalisis efisiensi dari penggunaan *Merkle Tree* dalam verifikasi integritas *chunk file* pada simulasi sistem penyimpanan terdistribusi. Simulasi dilakukan dengan membagi *file* menjadi beberapa *chunk* lalu menghitung nilai *hash* setiap *chunk* menggunakan algoritma SHA-256, membuat struktur data *Merkle Tree*, dan membandingkan waktu yang dibutuhkan dalam proses verifikasi integritas data dengan metode konvensional dengan proses verifikasi integritas data dengan *Merkle Tree*. Analisis difokuskan pada kompleksitas waktu, jumlah data yang perlu diverifikasi, dan ukuran data.

Keywords—*Merkle Tree*, *SHA-256*, *Binary Tree*, *Sistem Penyimpanan Terdistribusi*, *Integritas Data*.

I. PENDAHULUAN

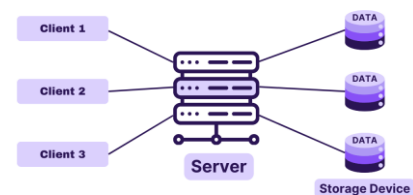
Perkembangan teknologi informasi telah memungkinkan penggunaan sistem penyimpanan terdistribusi (*distributed storage system*) dalam berbagai layanan digital, terutama *cloud storage*, sistem berkas terdistribusi, dan teknologi blockchain. Sistem penyimpanan terdistribusi memungkinkan data untuk disimpan dalam beberapa simpul (*node*) yang berbeda tetapi saling terhubung sehingga dapat meningkatkan skalabilitas, ketersediaan layanan, dan juga toleransi terhadap kegagalan sistem. Seiring bertambahnya ukuran data yang disimpan, dibutuhkan suatu mekanisme yang menjamin integritas data.

Integritas data sendiri merupakan jaminan bahwa data tidak akan berubah selama proses penyimpanan. Salah satu mekanisme yang umum digunakan pada *distributed storage system* adalah membagi sebuah *file* akan menjadi beberapa *chunk* dan akan didistribusikan ke berbagai *node* penyimpanan. Pendekatan ini meningkatkan efisiensi, tetapi ada tantangan baru

dalam proses verifikasi data karena apabila suatu *chunk* mengalami kerusakan, sistem harus dapat mendeteksi hal tersebut dengan cepat dan akurat.

Pendekatan verifikasi integritas secara konvensional umumnya dilakukan dengan menghitung kembali nilai *hash* dari keseluruhan berkas dan membandingkannya dengan nilai *hash* acuan. Mekanisme tersebut sederhana, tetapi biaya komputasinya meningkat seiring dengan bertambahnya jumlah *chunk* yang harus diperiksa. Pada sistem dengan volume data yang besar, proses verifikasi dapat menjadi salah satu sumber overhead yang memengaruhi efisiensi pengelolaan data.

Salah satu struktur data yang dapat digunakan untuk mengatasi hal tersebut secara efisien adalah *Merkle Tree*. Pada struktur data ini, setiap simpul daun (*leaf node*) akan menyimpan nilai *hash* dari suatu *chunk* data, sedangkan setiap simpul internal akan menyimpan nilai *hash* dari gabungan dua simpul anaknya. Seluruh data dalam sistem direpresentasikan dengan sebuah *root hash*. Dengan mekanisme ini, perubahan atau kerusakan pada suatu *chunk* hanya memengaruhi jalur dari simpul daun menuju *root* sehingga proses verifikasi integritas data tidak perlu dilakukan pada keseluruhan data.



Gambar 1. *Distributed Storage System*

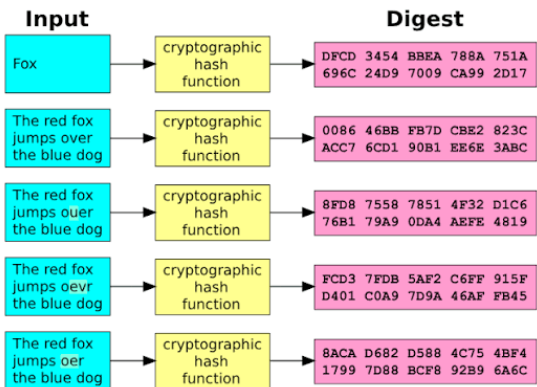
Sumber: Medium

II. LANDASAN TEORI

A. Cryptographic Hash Function and Secure Hash Algorithm

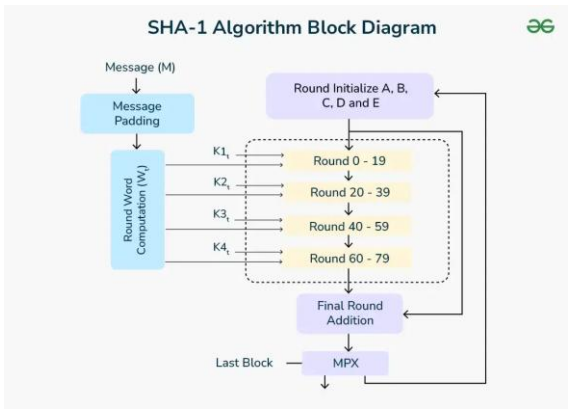
Cryptographic Hash Function atau fungsi *hash* kriptografis adalah menerima *input* data berukuran acak dan mengubahnya menjadi nilai *output* atau *hash* dengan ukuran yang tetap. Fungsi *hash* kriptografis dirancang agar proses perhitungan *hash*

menjadi lebih mudah untuk dilakukan, tetapi perubahan kecil pada data *input* dapat menghasilkan nilai *hash* yang sangat berbeda sehingga sering digunakan untuk menjaga integritas data. Fungsi *hash* kriptografis memiliki beberapa karakteristik yaitu sensitif terhadap perbedaan input, deterministic sehingga *input* yang sama akan selalu menghasilkan *hash* yang sama, bersifat *one-way* sehingga mustahil membuat data asli dari nilai *hash*-nya, dan sangat sulit untuk menemukan dua *input* berbeda yang menghasilkan *hash* yang sama.



Gambar 2.1 Cryptographic Hash Function
Sumber: ResearchGate

Secure Hash Algorithm terutama SHA-256 adalah sebuah algoritma berbasis fungsi *hash* yang dikembangkan oleh National Institute of Standards and Technology. Algoritma ini menerima *input* dengan panjang yang acak lalu menghasilkan nilai *hash* dengan Ppanjang konstan sebesar 256 bit. Sebelum pemrosesan, data *input* mengalami *padding* sehingga Panjang data memenuhi blok yang ditentukan. Data tersebut kemudian dibagi lagi menjadi blok berukuran 512 bit dan diproses melalui fungsi kompresi hingga menghasilkan nilai *hash* akhir.



Gambar 2.2 Secure Hash Algorithm
Sumber: GeeksForGeeks

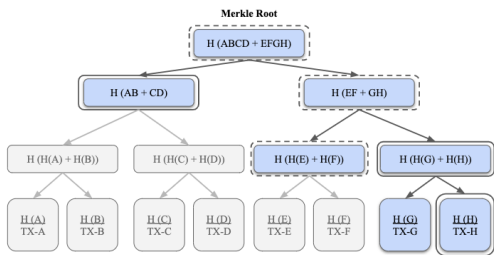
B. Merkle Tree as a Recursive Data Structure

Merkle Tree merupakan struktur data berbasis pohon biner (*binary tree*) rekursif yang diciptakan dan dipatenkan oleh Ralph Merkle pada tahun 1979. Pada pohon ini, setiap simpul daun merepresentasikan nilai *hash* dari satu blok atau *chunk* data. Sedangkan simpul dalam atau internal menyimpan nilai

hash dari gabungan nilai *hash* kedua anaknya. Simpul tertinggi dalam pohon Merkle disebut dengan *Root Hash* atau *Merkle Root*.

Secara matematis dan menggunakan algoritma SHA-256, jika sebuah simpul dalam N memiliki dua anak yaitu simpul kiri L dan simpul kanan R, nilai *hash* dari simpul N didefinisikan sebagai:

$$H(N) = SHA256(H(L) + H(R))$$



Gambar 2. Merkle Tree
Sumber: ResearchGate

C. Authentication Path atau Merkle Proof

Authentication path adalah himpunan *hash* saudara yang diperlukan untuk membuat root dari satu chunk tertentu. Dalam verifikasi, agar suatu sistem tidak memproses seluruh pohon, digunakan mekanisme ini supaya sistem hanya menghitung ulang hash chunk target lalu menggabungkannya dengan *hash* pada jalur autentikasi sampai mencapai *root*. Mekanisme ini menggeser verifikasi dari pemeriksaan keseluruhan menjadi hanya pemeriksaan satu jalur pohon.

D. Space and Time Complexity

Kompleksitas waktu atau *Time Complexity* adalah ukuran waktu komputasi yang dibutuhkan oleh suatu program untuk menyelesaikan masalah atau melakukan operasi. Sedangkan kompleksitas ruang atau *Space Complexity* adalah jumlah memori yang dibutuhkan oleh suatu program untuk menyelesaikan masalah atau melakukan operasi. Keduanya berjalan seiring bertambahnya *input* yang sering dilambangkan dengan *n*.

Dalam konteks kompleksitas waktu, ada istilah yang sering dikenal dengan *Big-O notation*. Notasi ini melambangkan efisiensi dari suatu program atau algoritma. Berikut adalah tabel dari representasi waktu dalam notasi *Big-O* yang umum digunakan dan diurutkan dari yang paling efisien.

O(1)	Waktu akan selalu sama tidak peduli jumlah <i>input</i> .
O(log n)	Waktu akan bertambah semakin lambat meski <i>input</i> -nya bertambah.
O(n)	Waktu akan berbanding lurus dengan jumlah <i>input</i> .
O(n log n)	Waktu akan berbanding lurus dengan jumlah <i>input</i> sekaligus ditambah lagi meski akan bertambah semakin lambat.
O(n ²)	Waktu kuadratik dari jumlah <i>input</i> -nya

$O(n^3)$	Waktu kubik atau pangkat tiga dari jumlah <i>input</i> -nya
$O(2^n)$	Waktu akan berlipat ganda setiap kali ada <i>input</i> .
$O(n!)$	Waktu akan menghitung factorial dari jumlah <i>input</i> .

III. METODOLOGI

Simulasi ini direpresentasikan sebagai kumpulan *chunk file* yang berukuran tetap. Setiap *chunk* dipetakan ke SHA-256 untuk membuat *leaf hash*, lalu pasangan *hash* digabung-gabung hingga pada akhirnya membentuk suatu *root hash*. Jika pada suatu kondisi terdapat jumlah *leaf hash*-nya ganjil, maka *leaf hash* tersebut akan di duplikasi sehingga pohonnya tetap seimbang.

Implementasi dilakukan dengan menggunakan Python 3 dan library bawaan hashlib. Simulasi tidak berfokus pada hal-hal lain selain waktu dan biaya komputasi yang dibutuhkan dalam proses verifikasi.

Simulasi yang dilakukan memiliki alur sebagai berikut:

1. *Chunking*: berkas yang diberikan sebagai *input* dibagi menjadi beberapa *chunk* dengan ukuran yang sama.
2. *Hashing Dasar*: program memproses setiap *chunk* melalui fungsi atau lagoritma SHA-256 untuk membuat array berisi n buah *leaf nodes* atau *leaf hash*.
3. *Konstruksi Merkle Tree*: pasangan array daun dipasang-pasangkan atau dikonkatenasi, lalu di-*hash* lagi untuk membuat simpul dalam sampai terbentuk *Merkle Root* atau *Root Hash*.
4. *Simulasi Gangguan/Kerusakan (Tampering)*: program akan memanipulasi 1 byte data pada salah satu *chunk* untuk mensimulasikan kerusakan.
5. *Verifikasi dan Pembandingan*: akan dilakukan proses verifikasi integritas pada *chunk file*. Waktu eksekusi dihitung menggunakan fungsi `time()` di python untuk membandingkan kecepatan antara metode konvensional dan *Merkle Tree*.

A. Parameter Simulasi

Pengujian menggunakan 8 data, yaitu 8 sampai 1024 *chunk* dengan rasio dua kali lipat pada setiap langkah. Setiap *chunk* memiliki ukuran 1024 *byte* agar simulasi tetap sebanding, seimbang, dan tidak terlalu panjang. Alasan pemilihan data yang berkelipatan 8 adalah supaya pohon yang terbentuk seimbang sehingga memudahkan analisis dan simulasi.

B. Algoritma Pembuatan Merkle Tree

Setiap *chunk file* memiliki ukuran data yang seragam, yaitu sebesar 1024 *byte* atau 1 KB. Dengan demikian, ukuran total berkas yang disimulasikan akan bertambah dari 8 KB hingga 1024 KB seiring bertambahnya jumlah *chunk*.

Proses konstruksi pohon dilakukan secara *bottom-up*. Langkah pertama adalah memetakan seluruh elemen *chunk*

menjadi himpunan nilai *hash* awal menggunakan algoritma SHA-256 sebagai fungsi standar. Berikut adalah implementasi fungsi konstruksi struktur data *Merkle Tree* dalam bahasa Python:

```
def build_merkle_tree(chunks: list) → list:
    leaves = [sha256(chunk) for chunk in chunks]
    tree = [leaves]

    while len(tree[-1]) > 1:
        current_level = tree[-1]
        next_level = []

        if len(current_level) % 2 != 0:
            current_level.append(current_level[-1])

        for i in range(0, len(current_level), 2):
            left = current_level[i]
            right = current_level[i+1]

            parent_hash = sha256((left + right).encode('utf-8'))
            next_level.append(parent_hash)

        tree.append(next_level)

    return tree
```

Gambar 3.1 Algoritma Pembuatan Merkle Tree

Algoritma ini memiliki alur sebagai berikut:

1. *Inisialisasi Daun*: algoritma memetakan seluruh himpunan *chunk* data menjadi himpunan *leaf nodes* menggunakan fungsi SHA-256.
2. *Mekanisme Padding*: struktur *Merkle Tree* menggunakan struktur pohon biner penuh sehingga apabila jumlah simpulnya ganjil akan dilakukan duplikasi simpul terakhir dan menambahkannya ke level yang sama.
3. *Iterasi Konkatenasi*: algoritma menggabungkan dua simpul yang saling bersaudara atau bertetangga, menggabungkan nilai *hash* dari keduanya, dan menghitung nilai *hash* baru dari gabungan tersebut sehingga membentuk suatu simpul dalam. Proses ini akan terus di-*looping* sampai terbentuk *Merkle Root*.

C. Algoritma untuk Authentication Path

Untuk melakukan proses verifikasi, sistem tidak memerlukan seluruh simpul yang terdapat dalam pohon biner tersebut, sistem hanya memerlukan suatu jalur bukti atau bisa disebut *authentication path*.

```
def generate_merkle_proof(tree: list, target_index: int) → list:
    proof = []
    idx = target_index
    for level in tree[:-1]:
        if idx % 2 == 0:
            if idx + 1 < len(level):
                proof.append(level[idx + 1])
            else:
                proof.append(level[idx - 1])
            idx //= 2
    return proof
```

Gambar 3.2 Algoritma Authentication Path

Algoritma ini memiliki alur sebagai berikut:

1. Penelusuran jalur/*path*: algoritma ini dari posisi indeks *chunk* yang ditargetkan hingga ke bawah *root*, menyimpan nilai *hash* dari simpul saudara.
2. Penentuan Posisi: Operasi matematika modulo digunakan untuk menentukan posisi saudara. Jika indeks saat ini genap, berarti simpul berada di posisi cabang kiri, sehingga saudaranya adalah simpul di sebelah kanannya. Sebaliknya, jika ganjil, simpul berada di posisi kanan, sehingga saudaranya ada di sebelah kiri.

D. Algoritma Verifikasi Chunk File

Algoritma utama yang memverifikasi apakah *chunk* mengalami kerusakan atau tidak.

```
def verify_chunk(target_chunk: bytes, target_index: int, proof: list, trusted_root: str) -> bool:
    current_hash = sha256(target_chunk)
    idx = target_index
    for sibling_hash in proof:
        if idx % 2 == 0:
            current_hash = sha256((current_hash + sibling_hash).encode('utf-8'))
        else:
            current_hash = sha256((sibling_hash + current_hash).encode('utf-8'))
        idx //= 2
    return current_hash == trusted_root
```

Gambar 3.3 Algoritma Verifikasi Chunk File

Algoritma ini memiliki alur sebagai berikut:

1. Komputasi Parsial: Fungsi ini menghitung *hash* dari *chunk* yang dicurigai atau ingin diverifikasi, kemudian membuat jalur *hash* ke atas menggunakan himpunan jalur bukti yang sebelumnya dibuat.
2. Sensitivitas Urutan: Penggabungan tidak bersifat komutatif (tidak seperti operasi tambah/kali) sehingga algoritma perlu memastikan urutannya. Jika indeksnya ganjil maka *hash* saat ini digabung dibelakang *hash* saudara (*hash_saudara* + *hash_sekarang*). Jika indeksnya genap maka *hash* saat ini digabung di depan *hash* saudara.
3. Validasi Akhir: Hasil komputasi akhir pada level tertinggi akan dibandingkan dengan nilai *Merkle Root* tepercaya (*trusted_root*). Jika bernilai sama (*True*), data dipastikan utuh. Jika berbeda (*False*), data dipastikan telah rusak atau dimanipulasi.

E. Algoritma Secure Hash Algorithm dan Penggunaan Library

```
import hashlib
import time
import random

def sha256(data: bytes) -> str:
    return hashlib.sha256(data).hexdigest()
```

Gambar 3.4 Algoritma SHA-256 dan Library

Fungsi SHA-256 bekerja dengan cara menerima menerima input berupa data biner (*bytes*) dan mengembalikan nilai *hash* dalam format *string* heksadesimal sepanjang 64 karakter.

F. Algoritma Verifikasi Chunk File

```
if __name__ == "__main__":
    CHUNK_SIZE = 1024
    chunk_counts = [8, 16, 32, 64, 128, 256, 512, 1024]

    for n in chunk_counts:
        simulated_chunks = [random.randbytes(CHUNK_SIZE) for _ in range(n)]

        full_tree = build_merkle_tree(simulated_chunks)
        trusted_root = full_tree[-1][0]
        trusted_hash_conv = build_conventional_hash(simulated_chunks)

        target_idx = random.randint(0, n - 1)
        original_chunk = simulated_chunks[target_idx]
        proof = generate_merkle_proof(full_tree, target_idx)

        start_c = time.perf_counter()
        verify_conventional(simulated_chunks, trusted_hash_conv)
        end_c = time.perf_counter()

        start_m = time.perf_counter()
        is_valid_normal = verify_chunk(original_chunk, target_idx, proof, trusted_root)
        end_m = time.perf_counter()

        corrupted_chunk = bytearray(original_chunk)
        corrupted_chunk[0] = (corrupted_chunk[0] + 1) % 256
        is_valid_corrupt = verify_chunk(bytes(corrupted_chunk), target_idx, proof, trusted_root)

        conv_ms = (end_c - start_c) * 1000
        merkle_ms = (end_m - start_m) * 1000
        status_asli = "Baik" if is_valid_normal else "Gagal"
        status_korup = "Korup" if not is_valid_corrupt else "Gagal"

        print(
            f"{n}<8} | "
            f"{status_asli:<12} | "
            f"{status_korup:<12} | "
            f"{conv_ms:>25.6f} | "
            f"{merkle_ms:>20.6f}"
        )
```

Gambar 3.5 Algoritma Pengujian

Gambar 3.5 merupakan fungsi main, fungsi utama yang akan menjalankan keseluruhan program. Fungsi menggunakan beberapa library bawaan Python seperti *time* untuk menguji perbedaan waktu antara metode konvensional dan metode dengan *Merkle Tree*.

IV. HASIL DAN ANALISIS

A. Hasil dan Analisis Teoritis

Dalam pemodelan sistem penyimpanan terdistribusi, struktur data *Merkle Tree* dapat direpresentasikan sebagai *Directed Acyclic Graph* (*DAG*) yang secara spesifik membentuk pohon biner penuh (*full binary tree*). Berdasarkan prinsip teori graf, jika sebuah berkas dipecah menjadi *chunk* sebanyak *n* daun (*leaf nodes*), maka total keseluruhan simpul *V* yang harus dialokasikan di dalam memori sistem dapat dihitung dengan persamaan:

$$|V| = 2n - 1$$

Karakteristik utama dari pohon biner yang seimbang ini adalah tinggi pohonnya bertumbuh secara logaritmik. Tinggi pohon ini secara langsung merepresentasikan panjang jalur bukti yang diperlukan untuk memverifikasi satu *chunk* tunggal tanpa harus memeriksa *chunk* lainnya.

Tabel berikut menunjukkan pertumbuhan panjang *Merkle Proof* atau *Authentication Path* pada pohon biner yang seimbang. Nilai ini dihitung dengan cara menggabungkan dua *chunk* sampai hanya terdapat satu *node* atau *Merkle Root*.

Jumlah <i>Chunk</i> (n)	Tinggi Pohon	Panjang <i>Merkle Proof</i>
8	3	3
16	4	4
32	5	5
64	6	6
128	7	7
256	8	8
512	9	9
1024	10	10

Tinggi pohon dan panjang *authentication path* juga bisa dihitung menggunakan rumus matematis berikut:

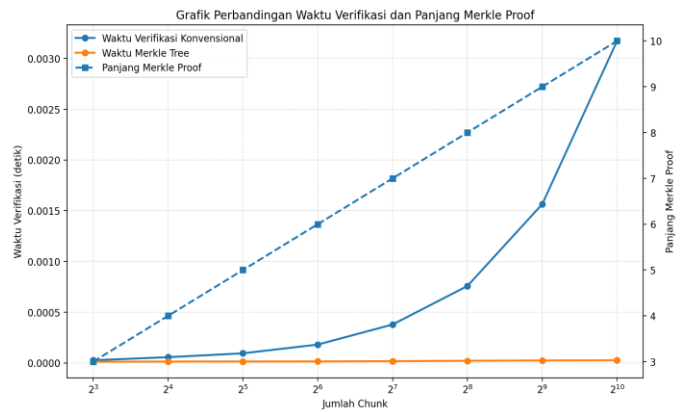
$$\text{Tinggi Pohon/Panjang Proof} = \log_2 n$$

Persamaan diatas menjadi bukti bahwa panjang proof atau *authentication path* meningkat secara logaritmik terhadap jumlah *chunk file*.

B. Hasil Benchmark Lokal Untuk Waktu

Menggunakan program Python 3, berikut adalah waktu yang dibutuhkan antara dua metode verifikasi integritas data, yaitu metode *Merkle Tree* dan metode konvensional (waktu yang tercatat merupakan waktu dalam detik):

Jumlah <i>Chunk</i>	Panjang <i>Proof</i>	Waktu Verifikasi Konvensional	Waktu <i>Merkle Tree</i>
8	3	0.00002670	0.00001240
16	4	0.00005650	0.00001390
32	5	0.00009460	0.00001450
64	6	0.00018090	0.00001480
128	7	0.00037840	0.00001770
256	8	0.00075790	0.00002140
512	9	0.00156450	0.00002430
1024	10	0.00317310	0.00002600



Grafik 4.1 Perbandingan Waktu Verifikasi
Sumber: Python Matplotlib

Berdasarkan hasil diatas, bisa dilihat kalau laju perubahan waktu jika menggunakan metode konvensional memiliki rasio kurang lebih dua. Sehingga, untuk setiap penambahan jumlah *chunk* dengan rasio dua kali lipat dari sebelumnya, waktu yang dibutuhkan juga meningkat dengan rasio yang hampir sama dengan rasio penambahan *chunk*. Jika kita membandingkan waktu yang dibutuhkan untuk jumlah *chunk* = 8 dan jumlah *chunk* = 1024, terlihat bahwa dibutuhkan sekitar 0.00002670 detik untuk memverifikasi integritas data pada jumlah *chunk file* = 8, tetapi dibutuhkan waktu sebesar 0.00317310 detik untuk memverifikasi integritas data pada jumlah *chunk file* = 1024. Terlihat disini bahwa perubahan waktunya cukup signifikan.

Namun, untuk laju perubahan waktu jika menggunakan struktur data *Merkle Tree*, laju perubahan waktunya sangat tipis, meskipun jumlah *chunk file* ditambah dengan rasio dua kali lipat, perubahan waktunya meningkat dengan margin yang sangat kecil. Bisa dilihat pada jumlah *chunk* = 8, waktu yang dibutuhkan adalah sekitar 0.00001240 detik. Saat jumlah *chunk* ditingkatkan sampai 1024, waktu yang dibutuhkan meningkat dari 0.00001240 detik menjadi 0.00002600 detik. Sehingga kesimpulannya, hanya butuh 0.00001360 detik tambahan untuk memverifikasi 1024 *chunk file* dibandingkan dengan 8 *chunk file*.

Berdasarkan analisis diatas, kompleksitas waktu yang dibutuhkan untuk memproses atau memverifikasi integritas secara konvensional adalah sebesar $O(n)$ sedangkan waktu yang dibutuhkan untuk memverifikasi integritas data menggunakan struktur data *Merkle Tree* adalah sebesar $O(\log n)$. Dengan membandingkan kedua metode diatas, terlihat bahwa *Merkle Tree* menghabiskan waktu yang lebih sedikit ketika dibandingkan dengan metode konvensional, yaitu menelusuri tiap data.

C. Hasil Benchmark Lokal untuk Ruang

Dari segi kompleksitas ruang, verifikasi integritas data dengan metode konvensional dilakukan dengan menelusuri setiap *chunk file* dan mengekstrak n *chunk file*. Sehingga pada jumlah *chunk file* = 1024, dengan ukuran masing-masing *chunk* adalah sebesar 1 KB, sistem perlu mengevaluasi sebesar 1 MB. Sedangkan jika dibandingkan dengan struktur data *Merkle*

Tree, secara matematis kita bisa menghitungnya sebagai berikut:

$$\text{Jumlah Chunk File} = 1024 = 2^{10}$$

$$\text{Panjang Proof Merkle} \log_2 2^{10} = 10$$

$$\text{Ukuran output SHA256} = 32 \text{ byte}$$

$$\text{Ukuran yang perlu dievaluasi} = 32 * 10 = 320 \text{ byte}$$

Algoritma SHA-256 membuat semua ukuran input data menjadi 32 byte, sehingga dengan menggunakan algoritma SHA-256 dan struktur data *Merkle Tree*, sistem hanya perlu menganalisis atau memverifikasi sebesar 320 byte.

V. KESIMPULAN

Struktur data *Merkle Tree* yang diintegrasikan dengan algoritma SHA-256 terbukti secara signifikan menghemat waktu dan ruang memori. Metode konvensional juga terbukti menghabiskan waktu dan memori yang jumlahnya semakin signifikan seiring bertambahnya *chunk file*. Dengan mengimplementasikan *Merkle Tree* dan SHA-256, proses verifikasi integritas data dapat berlangsung dengan lebih cepat tanpa perlu mengeluarkan biaya, memori, dan waktu yang banyak.

UCAPAN TERIMA KASIH

Saya ingin mengucapkan rasa syukur yang sebesar-besarnya pada Tuhan yang Maha Esa atas anugerah dan berkatnya sehingga makalah ini dapat diselesaikan dengan baik. Saya juga ingin mengucapkan terima kasih pada Bapak Dr. Ir. Rinaldi Munir, M.T., selaku dosen matematika diskrit atas bimbingan dan wawasan yang telah diberikan selama satu semester ini.

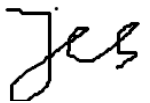
REFERENSI

- [1] [1] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*, Springer, 1988.
- [2] National Institute of Standards and Technology, *FIPS PUB 180-4: Secure Hash Standard (SHS)*, 2015.
- [3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
- [4] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed., Pearson, 2007.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
- [6] IETF, *RFC 9162: Certificate Transparency Version 2.0*, 2023.
- [7] Rosen, K. H. (2012). *Discrete Mathematics and Its Applications* (7th ed.). McGraw-Hill.
- [8] Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice* (7th ed.). Pearson.

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



Justin William
13525042