Investigating practical optimisations for data verification

How can the block sizes of a Merkle tree's leaf nodes be optimised as a function of the unreliability of the channel to maximise speed in data verification?

Computer Science Extended Essay

Word count: 3879

Contents

1 Introduction **3**

2 Theoretical Background **4**

2.1 Hash Functions 4

2.2 Merkle Trees 6

2.3 Network Unreliability 7

3 Primary Data **9**

3.1 Hypothesis 9

3.2 Methodology 10

3.3 Observations 14

3.4 Results 15

4 Data Analysis **18**

5 Next Steps **20**

6 Conclusion **21**

Appendix **22**

Bibliography **26**

1 Introduction

Data verification is integral to any process involving data transfer. It is the process by which a user can verify the data they received is the same as what they were sent. Whether verifying data from network requests or locally transferring files, modern computer users make use of this process in almost every application. There are a myriad of methods to verify data, ranging significantly in complexity based on both the data and how strict the verification needs to be. Speed is the primary factor when considering methods of data verification, this is apparent in that contemporary data verification is so seamlessly quick users are often unaware it even takes place. In this paper, only strict equality in terms of data verification will be discussed, where data is verified exactly and no deviation is tolerated. This paper will be centred around data verification, particularly for transmission across the Internet, with consideration of the various variables not present in local transmission.

The most straightforward way to verify a downloaded file is by comparing it directly with itself, essentially iterating through the data and seeing if each value matches up. Of course, this raises a number of issues in terms of both speed and efficiency, however in the context of downloading a file, issues such as visibility and network corruption arise. A cryptographic hash solves both of these issues, performing mathematical operations on data and outputting an indecipherable hash output of fixed length. The size of the fixed output will also generally be much smaller than the size of the file being downloaded saving time in large downloads. This method of transfer and verification is not sustainable as the sizes increase, however: this is because if data is corrupted, the entire file will have to be re-downloaded. Thus, for the transmission of large files, it'll first be split into chunks. These chunks additionally

help narrow down the source of corruption ensuring only corrupted data chunks are re-downloaded rather than the entire file itself. In order to increase efficiency and organisation, the chunks also known as leaves are grouped through merkle trees. This paper seeks to optimise the size of these leaves with regard to the rate of corruption in the medium of transmission to minimise the time taken to verify the data.

2 Theoretical Background

2.1 Hash Functions

“Hash functions are simply functions that take inputs of some length and compress them into short, fixed-length outputs.” (Katz & Lindell, 2021). The semantics of what is entailed by compressing the inputs is mostly irrelevant to the investigation however a brief understanding is necessary to prove the security of the hash function that will be used in this investigation. SHA-256 is a hash function a part of the SHA-2 family, designed by the NSA and first published in 2001 (Penard & van Werkhoven, 2008). The hash, also known as a digest, generated from hash functions is used generally to determine if the input has changed since the hash was generated. Essentially, for any given input into a hash function, a unique identifier is outputted, notably of fixed size, hence decreasing the time taken in data transfer. In our case this unique identifier is used to determine whether or not an input has changed when being transmitted across a channel, in an O(n) or a fixed 256 bit (SHA256) manner.

The National Institute of Standards and Technology (NIST) has a publication on Secure Hash Standards (SHS) for various hashing algorithms. There are two main stages that take place in the SHA-256 algorithm, preprocessing and hash

computation. The message is first converted into binary before being padded with bits to reach a multiple of 512, once padded it is parsed into message blocks now called the initial hash value.

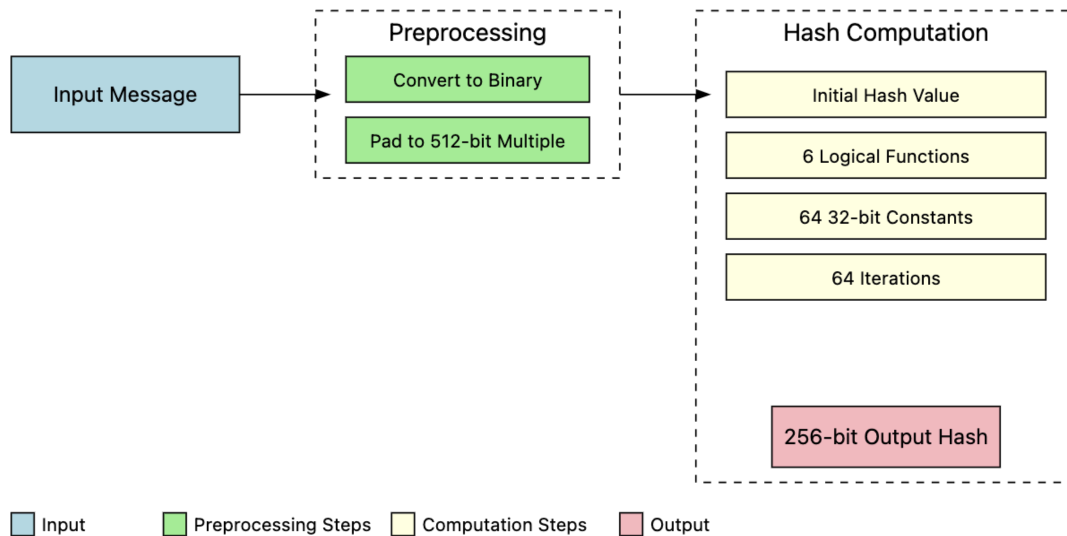


Figure 1: Hashing procedure

After being processed, an output is computed through the use of 6 logical functions such as AND, OR and XOR as well as 64 32-bit constants. The output becomes the input for the next iteration of which there is a total of 64. Included in the appendix are details on the 6 logical functions and 64 32-bit constants as well as the hash computation instructions; it is important to note however that understanding the inner workings of hash functions is entirely unnecessary to understand its applications. Essentially, the iterative operations in SHA-256 introduce added complexity and randomness making it more resistant to cryptographic attacks that may have threatened earlier or simpler hash algorithms.

2.2 Merkle Trees

Merkle trees are a space efficient data structure that makes it easy to verify data. A large file can be distilled into the fixed size of the hashing algorithm being used: in our case, SHA256 hashes are 256 bits long (Merkle, 1987). As previously explained in the hash functions section, any change or corruption of any file would cause the root hash to be distinctly different, allowing quick identification of a data verification error by verifying a fixed amount of data as opposed to a variable amount. In addition, the ‘tree’ format of the data allows a traversal such that the specific file that was corrupted can be identified using the merkle proof. Merkle proofs contain the hashes of various parts of the tree such that when verified against a specific hash the full merkle tree and thus the roots can be compared. Merkle proofs are thus a space efficient way to identify errors whilst also limiting the amount of data transferred thus decreasing the time taken to transfer the verification of the data. Hence, merkle trees are a widely used data structure in applications where data verification is essential (Becker, 2008).

Merkle trees are built upon hash functions. A merkle tree is constructed by taking partitions, in our case the even (2^n) splitting of a file, and hashing each partition that makes up the leaves of the tree. The hashes of these partitions are then hashed together and those hashes again hashed together such that it forms a root hash like seen below.

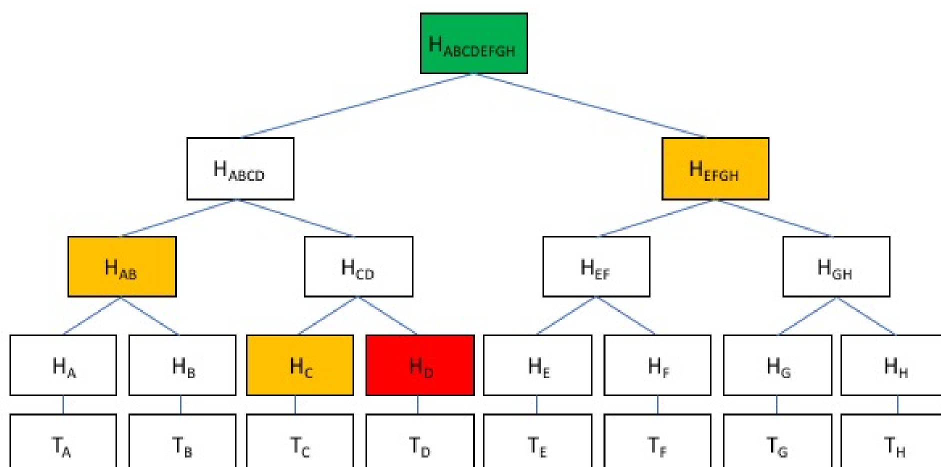


Figure 2: Corrupted file in a merkle tree

It is helpful to realise that the main purpose of hash functions to us and this experiment is to determine whether or not something has changed. By hashing and combining these different file partitions we are able to determine which specific parts of the original file have been corrupted during transmission. As seen in Figure 2, the yellow nodes are acceptable nodes whose children have not been affected by the corruption of any files. The right side of the tree is entirely unaffected by corruption, in an average or best case scenario when identifying files for requery, an O(logn) amount of nodes would need to be checked and queried for again, as opposed to O(n) nodes amount in a structure of e.g. an array (Cachin & Camenisch, 2004).

2.3 Network Unreliability

Network reliability is the measure of the number of bits transferred compared to the number of bits received by the dress without corruption. This is known as a bit error rate and for modern LANs should not exceed one error in 10 billion bits transmitted (Elliott, B. J., 2000). Our experiment is thus catered towards more extreme cases

whereby networks are severely impaired these include but are not limited to: times of high network congestion, topological obstructions, and single points of failure in decentralised networks. These scenarios best illustrate the efficacy of the data structure however it is certainly applicable to other situations.

There are two main ways to simulate network unreliability, the first way is by actually sending the data through a channel and purposefully corrupting a certain number of the files being sent. Another is to make certain assumptions and consider the network's reliability while standardising other elements of the data transfer such as requeries for corrupted data, speed of data transmission and computation.

When a file partition from the merkle tree is sent, an accompanying merkle proof is sent to verify against the file to see if it has been corrupted during transfer. There are a number of complexities regarding network reliability that can be overlooked in this situation of ideal transfer, for example any data sent over the network, not just limited to the sent file but also the accompanying merkle proof is subject to corruption. Thus when considering the speed at which the network transfer takes place, requeries for the corrupted data need to consider both the network speed, size and validity of the proofs as well as file sizes when accounting for the simulated time.

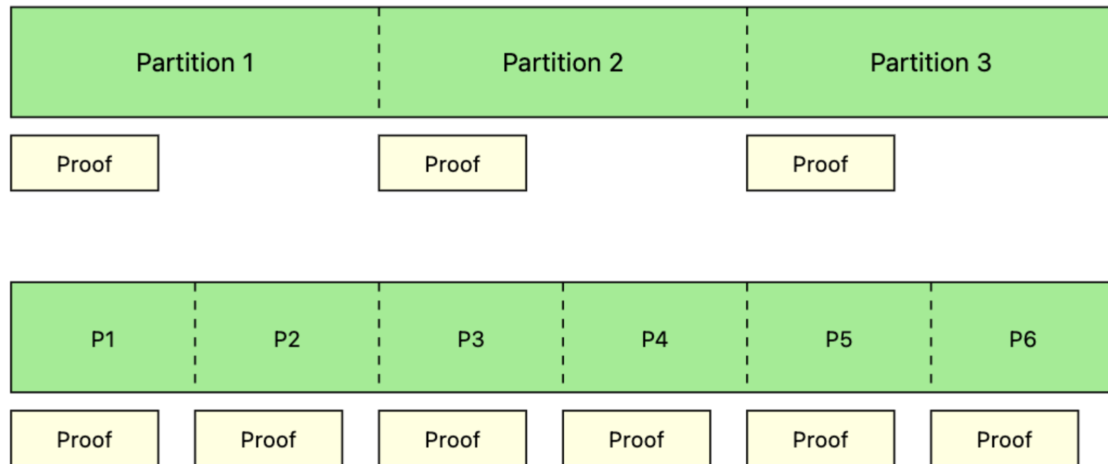


Figure 3: Increasing total transmitted data with fixed proofs

3 Primary Data

3.1 Hypothesis

A large file cannot risk transmission in an unreliable network as the corruption of any part of transmission would require the retransmission of the entire file. By splitting this file into different partitions, each partition can be checked to narrow down the root of the error; with a merkle tree the process of this checking is even more efficient, being able to determine if certain subtrees are accurate and thus reducing the total time in data verification. The main dilemma here is thus the number of partitions the merkle tree should be made up of. More partitions increase granularity in error detection but increases overhead in computation of the related merkle operations. Moreover, the benefits of the granularity in error detection diminish as the number of partitions increases while the computational costs, being based on SHA256, are fixed. Therefore, I hypothesise that there is an equilibrium between the

benefits of the merkle tree's efficiency in finding errors that still respects the fixed overhead involved in a higher number of partitions, to result in the lowest amount of time.

3.2 Methodology

Our primary goal is to determine whether or not there is an optimal file size for leaves of a merkle tree in verifying data transfer across an unreliable channel. The case is that, for larger leaf sizes or fewer partitions, corruption in transmission would result in the entire file or partition having to be retransmitted, increasing the total amount of time required to transfer the file, whilst accounting for any unreliability. On the other hand, having too many partitions or leaf sizes too small would similarly result in an increase in the amount of time required to correctly transfer the file due to the process of hashing and constructing the larger tree.

A program can be used to simulate and test the variance in the size of the merkle tree's leaf sizes and the time taken to transfer these files. The program can be split into three parts, the setup of the experiment, the core algorithm simulating the verification and requerying of data, and the testing of it. To set up the experiment, we can first model the file being transferred, e.g. a photo. This photo is then converted into base64, converting the binary values of the image into printable characters which can then be made into partitions of 2^x to match a perfect merkle tree. Some complexities involved in the cleaning up of the data and converting the image to text to construct the tree are also necessary and included in the code in the appendix however it is secondary to the essence of the experiment, the data verification, and

will thus be omitted for clarity. A merkle tree can be created from these partitions, hashing each file in succession as explained in Section 2.2.

```
const bs64_tides = fs.readFileSync("base64_tides.txt", "utf8");

function partitionString(string: string, numPartitions: number): string[] {
    if (numPartitions <= 0) return ["Invalid number of partitions"];
    const partitionSize = Math.ceil(string.length / numPartitions);
    const partitions: string[] = [];
    for (let i = 0; i < string.length; i += partitionSize) {
        partitions.push(string.substring(i, i + partitionSize));
    }
    if (partitions.length > numPartitions) {
        partitions[partitions.length - 2] += partitions.pop();
    }
    return partitions;
}
```

Figure 4: Data setup

Now that our inputs are set up we can simulate the amount of time expended by a given transfer of data, to this end, a few factors need to be identified. The main factors contributing to the time taken are network speed, transfer of the merkle proof, hashing speed, and merkle tree construction. The time taken to transfer the actual file and its accompanying merkle proof used to verify its validity can be calculated through the use of a few benchmarks.

By taking our photo as a benchmark, when deconstructed into characters it contains around 333,000, equivalent to around 333KB. We'll assume all network transfers take place at an average of 1 MB/s. Note, the specific benchmarks, file sizes and their real world validity are irrelevant as it is only the relative relationship between the merkle trees' variations and network transfers that we're finding.

$$1MB/s \; = \; \frac{0.333MB}{t}$$

$$t \; = \; 0.333s$$

Thus for a given block size of x characters, the time taken to send across a theoretical network can be found by multiplying x by 1×10^{-6}.

To fully simulate our data transfer and verification scenario, we'll need to consider the speed of hashing and merkle tree construction in addition to the speed of the network. We can take a few baselines such as the speed at which hashes can be generated, at 500MB/s, which I found by taking the average of some timings hashing files locally with SHA256. Hence we can calculate the simulated amount of time added by hashes and merkle tree construction using these baselines, and adding it to a simulated time variable. The summation of this and the actual run time representing the total time taken in data transmission and verification.

With the data to be transferred we'll now need to mimic network unreliability in a real-world setting. This can be done in several ways as mentioned in Section 2.3 however for our experiment we'll simulate it through the use of randomness. To test our three different channels of unreliability where 33%, 66%, and 99% of data files corrupt, a variable of a random value from 0 to 1 can be used. The process of data transfer being looped if the random value generated is e.g. smaller than or equal to 0.33, the simulated hashing and transfer of the leaf is calculated as shown above, thus mimicking the additional time taken in retransmission of corrupted files.

In addition to this, we will limit the amount of retries for corrupted data. This may introduce a bias towards more unreliable networks however it replicates the time out functionality of most modern query systems in order to preserve user experience and prevent indefinite waiting from the network's unreliability. This results in the following:

```
function requeryCorrupted(
    tree: MerkleTree,
    corruptedPartitions: string[],
    originalRoot: string,
    unreliability: number
): { root: string; time: number } {
    const leaves = tree.getLeaves();
    let localTime = 0;

    for (let i = 0; i < corruptedPartitions.length; i++) {
        const leaf = corruptedPartitions[i];
        const proof = tree.getProof(leaves[i]);
        const isValidLeaf = tree.verify(proof, leaf, originalRoot);

        if (!isValidLeaf) {
            let attempts = 0;
            while (attempts < 5) {
                localTime += (leaf.length + JSON.stringify(proof).length) * charSim;
                if (Math.random() >= unreliability) {
                    corruptedPartitions[i] = leaves[i].toString();
                    break;
                }
                attempts++;
            }
            // Add penalty for failed transmission, but make it less severe for smaller partitions
            if (attempts === 5) {
                localTime += leaf.length * charSim * 5 / Math.sqrt(corruptedPartitions.length);
            }
        }
    }
```

Figure 5: Corruption detection and requery

By combining the two functions, we can test variations in unreliability and etc:

```
function testingPowers(unreliability: number): any[] {
    let results: any = [];
    for (let i = 1; i <= 12; i++) {
        const numPartitions = Math.pow(2, i);
        const partitions = partitionString(bs64_tides, numPartitions);
        const tree = new MerkleTree(partitions, SHA256);
        const corruptedPartitions = partitions.map(x => mimicNetworkUnreliability(x, unreliability));
        const root = tree.getRoot().toString("hex");

        const start = performance.now();
        const { root: finalRoot, time: localTime } = requeryCorrupted(tree, corruptedPartitions, root, unreliability);
        const end = performance.now();

        const baseTime = (bs64_tides.length * charSim) / Math.sqrt(numPartitions);
        const totalTime = (end - start) / 1000 + localTime + baseTime;
        results.push({ partitions: numPartitions, time: totalTime });
        console.log(`${totalTime.toFixed(3)}`);
    }
    return results;
}
```

Figure 6: Test and results generation

3.3 Observations

Initially, I tried to find a shortcut by simulating hashes based on reliability. Essentially, I assumed that the bulk of the time added to compute was from the computer's processing of hashes and construction of the merkle tree. While this certainly adds time, it is much more fixed and thus would not scale with the implicated differences in file partitions and thus sizes being transmitted and retransmitted in case of corruption. Additionally, this is fundamentally wrong and doesn't utilise the unique property of merkle trees to not require traversing through the entire tree, simply using the merkle proof to identify specific subtrees that had errors. The fix was to shift the focus from calculating the time taken to hash and construct merkle trees to the simulated time taken as a result of the transfer of proofs and blocks based on size across the network. This process was described earlier, and the estimated network transfer speeds contributed to the majority of the computed time, serving as the variable time illustrating the time as opposed to the fixed time of the hashing computations.

When trying to find the corrupted leaf, simply verifying the proof against the original root does not work as the simulated network unreliability affects more than just the current leaf being verified, the corruption during transmission is across all leaves and thus the merkle proofs need to be sent with each file. Thus this issue can be fixed by querying for the same block and proof each time if the block transferred does not verify correctly with the sent proof. This issue is exacerbated by poor network

conditions and thus a retry cap is implemented to mimic real-world back off mechanisms waiting for network conditions to improve.

3.4 Results

Table 1: Total time taken vs number of partitions across a channel of 33% corruption

n	File Partitions (2^n)	Time Taken (s)					
		Trial 1	Trial 2	0.284	Trial 4	Trial 5	Average (s)
1	2	0.288	0.284	3.257	0.959	0.284	1.0144
2	4	1.229	0.89	1.182	0.886	0.55	0.9474
3	8	1.519	0.841	0.722	1.35	0.503	0.987
4	16	0.584	0.554	1.096	0.81	0.894	0.7876
5	32	0.75	0.665	0.755	0.838	0.794	0.7604
6	64	0.887	0.822	0.734	0.559	0.669	0.7342
7	128	0.908	0.837	0.805	0.999	0.86	0.8818
8	256	0.835	0.909	1.225	0.849	0.883	0.9402
9	512	1.281	1.371	1.865	1.135	1.081	1.3466
10	1024	1.897	1.741	3.087	1.877	1.787	2.0778
11	2048	3.007	3.038	5.678	3.083	3.086	3.5784
12	4096	6.027	6.274	6.019	6.019	5.854	6.0386

Graph 1: Graphed table from a range of 2-1024 partitions across a channel of 33% corruption

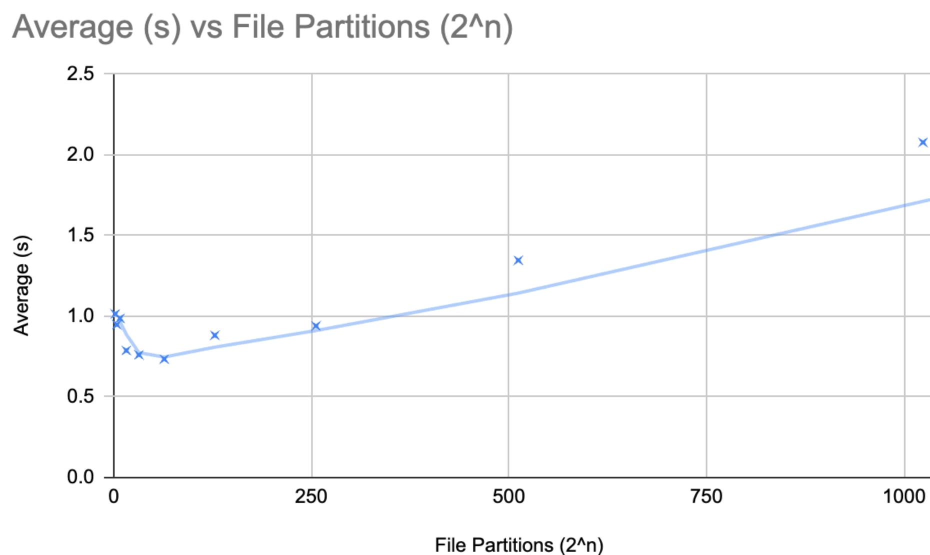


Table 2: Total time taken vs number of partitions across a channel of 66% corruption

n	File Partitions (2^n)	Time Taken (s)					
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average (s)
1	2	2.15	0.9	6.681	4.192	1.514	3.0874
2	4	1.139	2.758	2.141	2.38	1.759	2.0354
3	8	0.628	1.718	3.186	1.737	2.672	1.9882
4	16	2.233	2.412	2.075	2.868	1.852	2.288
5	32	2.174	2.006	2.514	2.366	2.011	2.2142
6	64	2.675	2.489	2.345	2.081	2.262	2.3704
7	128	2.395	2.276	2.189	2.147	2.457	2.2928
8	256	2.776	3.104	2.87	2.572	2.603	2.785
9	512	3.458	3.489	3.366	3.417	3.478	3.4416
10	1024	5.128	5.087	4.967	5.128	5.003	5.0626
11	2048	8.929	8.529	9.108	8.846	8.505	8.7834
12	4096	16.97	17.091	16.744	16.598	16.786	16.8378

Graph 2: Graphed table from a range of 2-1024 partitions across a channel of 66% reliability.

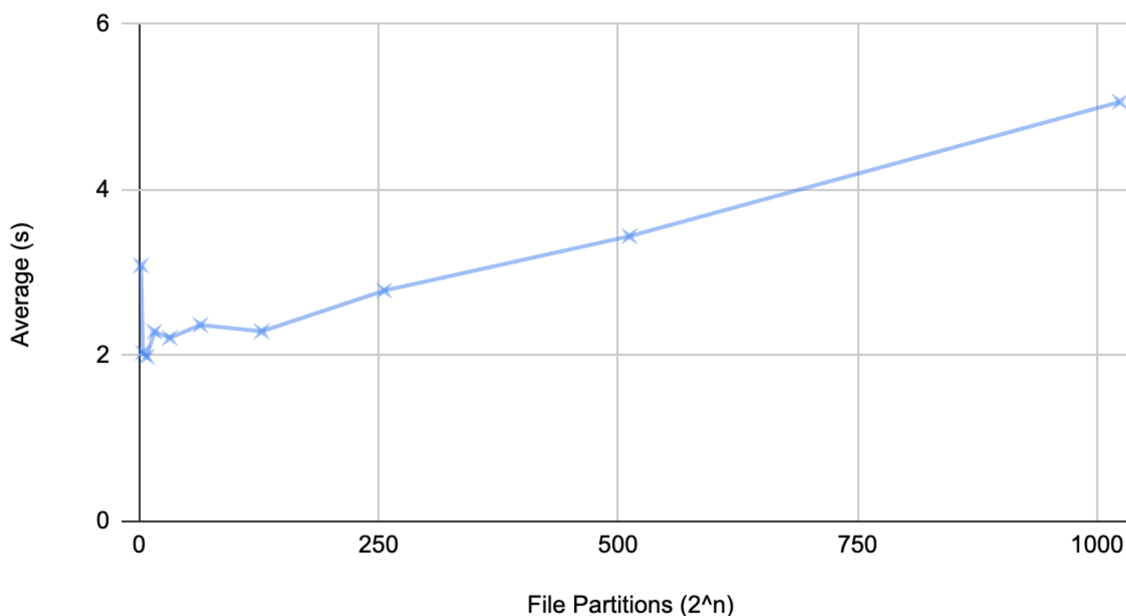
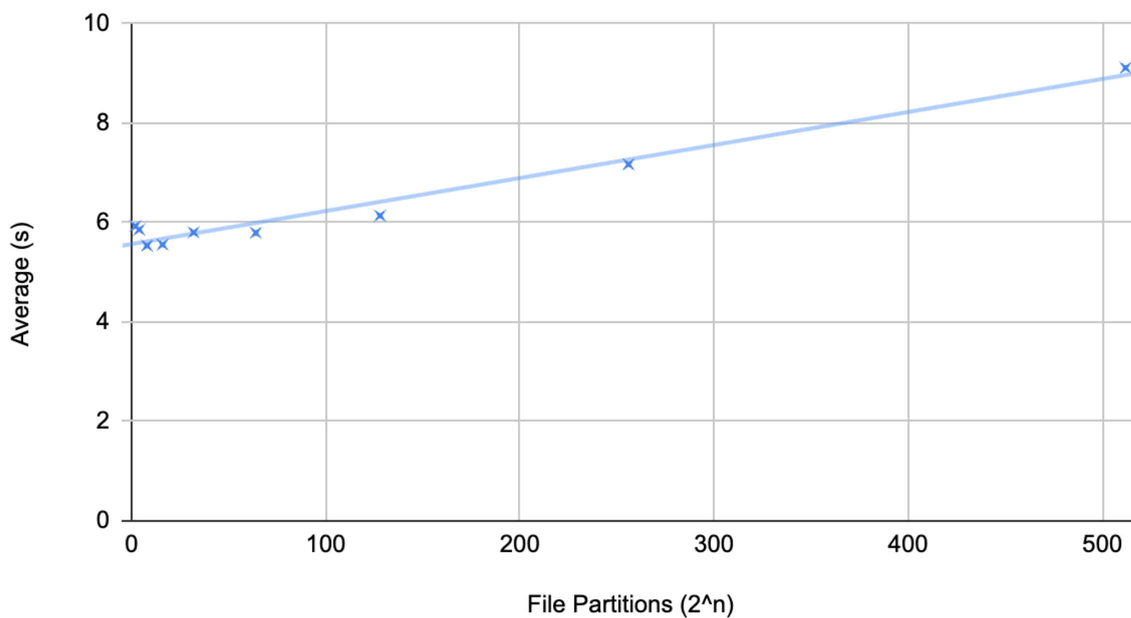


Table 3: Total time taken vs number of partitions across a channel of 99% corruption

n	File Partitions (2^n)	Time Taken (s)					
		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average (s)
1	2	5.929	5.927	5.927	5.927	5.928	5.9276
2	4	5.859	5.86	5.858	5.858	5.86	5.859
3	8	5.818	5.535	5.252	5.252	5.82	5.5354
4	16	5.742	5.46	5.386	5.386	5.812	5.5572
5	32	5.862	5.751	5.86	5.86	5.68	5.8026
6	64	5.672	5.776	5.78	5.78	5.962	5.794
7	128	6.184	6.022	6.081	6.081	6.326	6.1388
8	256	7.225	7.223	7.14	7.14	7.126	7.1708
9	512	9.082	9.116	9.105	9.105	9.154	9.1124
10	1024	13.683	13.651	13.552	13.552	13.597	13.607
11	2048	23.421	23.635	23.443	23.443	23.36	23.4604
12	4096	44.617	44.459	44.299	44.299	44.596	44.454

Graph 3: Graphed table from a range of 2-512 partitions across a channel of 99% corruption.



As there is a linear relationship and significant impact to what is shown on the graph when plotting the values above 1024 partitions, I've omitted the inclusion of points above 1024 partitions in the graph so as to better illustrate the relationship between the optimum number of partitions.

4 Data Analysis

As mentioned in 3.3 Observations, a retry cap may cause a systematic error in the results. It affects environments of high unreliability and large partition sizes, thus favouring smaller partitions in channels of higher unreliability. A more adaptive retry mechanism could be implemented in the future, adjusting the retry cap accordingly. Aside from this, there are not many experimental errors as the computations are fixed in all bit randomness when testing across unreliable channels. The only real improvements would be to increase complexity in areas such as varied network reliability which will be explored further in the next steps.

Across the channel with a 33% corruption rate, there is a clear U-like relationship seen between the first five 2^n partitions, the optimal number of partitions converging at around n = 6 or 64 partitions, though the adjacent points seem to be roughly the same in terms of time taken. This 'U' like relationship becomes less and less pronounced for the channel of 66% corruption rate which sees an optimum range of time in transmission at around 4 to 32 partitions before continuing the linear increase in time seen across both other channels after around 512 partitions. For the channel of 99% corruption rate there is an almost unnoticeable optimal number of partitions,

forming a linear relationship from about 64 partitions onwards, with the first 64 taking approximately the same amount of time on average.

The cause of the ‘U’ like relationship being less pronounced as the corruption rate of the channel increases: this can be attributed to two factors. With a lower rate of corruption, a merkle tree’s ability to distribute risk is far more effective as entire subsections of the tree can be disregarded, therefore more quickly narrowing down the corrupted partition for requery. In addition to this, as the corruption rate is higher, the fixed data such as the merkle proofs will also need to be retransmitted more and more, thus making this normally negligible factor to majorly contribute to the total time taken, neutering the obvious ‘U’ shape.

The initial point of two partitions, is sensibly quite high and also quite inconsistent in its spread of data as would be expected from the nature of its transmission. These large partitions have a higher probability of corruption in transmission and thus would require the retransmission of each partition. From around 256 to 4096 partitions there is a linear relationship, the time taken seems to increase proportionately with the number of partitions as the sizes of each partition decreases and the fixed times to transfer the partitions and their accompanying merkle proofs increase.

My hypothesis that an equilibrium would be reached between the partition sizes and total time taken seems to take place across all test cases. The valley like relationship of a convergence taking place particularly in the channels of lower corruption rate, reaching a point of equilibrium in time taken from 2 to 64 partitions whereby the time

taken increases proportionally as presumably the fixed sizes of the partitions, its merkle proofs, and any related computation increases.

5 Next Steps

As is, the experiment tests a few set channels of unreliability, from where 33%, 66%, and 99% of packets sent will be corrupted and require retransmission. These channels have provided clear benchmarks on the general relationship between channel unreliability and number of file partitions in order to create a more practical formula or solution, changing the specific reliability of the channel such that it is more analogue will be essential. The changes in reliability can be modelled either as random bursts of unreliability, or by linearly changing the percentage of corruption from 1 to 100% in 1% increments. This would enable the modelling of a mathematical function that graphed the optimal file partition size on average and across every amount of channel unreliability, making the experiment's results more practical for applications.

Extending upon the real world applications of this experiment, the bursts in channel reliability could be combated with an exponential backoff strategy for retries. This would essentially increase the duration between requeries of corrupt data assuming that the network should recover given enough time. This would be a highly practical experiment and based on the results of this experiment, more relevant to the lower amount of channel reliability that a normal network user would realistically face. By dynamically adapting the size of file partitions to most effectively utilise a merkle tree's ability to detect errors and requery, network requests across unreliable channels would be much more efficient, ready to be applied to real world scenarios.

6 Conclusion

This paper has investigated the relationship between the number of file partitions and the reliability of the channel. For channels with a low corruption rate there is a clear optimum number of partitions at around 2^6 or 64 partitions. This value was experimentally found and may thus differ in different conditions. It serves, however, as a clear indication of the existence of a standard number of partitions for the experimented channels of reliability. As the unreliability of the channel increases, this optimum number of partitions becomes less clear. I hypothesise that this is due to the fact that as the unreliability of the channel increases, the files are more likely to be corrupted on average, thus increasing both the overhead fixed hashing and merkle tree computations as well as increasing the variable number of requeries. The logarithmic efficiency of the error detection becomes void in channels with high unreliability. In these cases, the subsequent queries by the Merkle tree result in a more simple linear check of each hash's validity due to the increase in corruption.

To answer the research question, block sizes of a merkle tree can be optimised by splitting the file into 2^6 number of partitions, though this optimum varies slightly and the adjacent *n* number of partitions work similarly well in minimising the total amount of time involved in the verification of the data transmission. This optimum is less obvious in channels of higher unreliability however as a merkle tree's efficiency decreases significantly when most if not all the root blocks are corrupted, essentially defaulting to a worse version of the linear check of the block's hashes, with pointless transferred merkle proofs, hashing computations, and merkle tree construction.

Appendix

SHA-256 6 logical functions:

4.1.2 SHA-224 and SHA-256 Functions

SHA-224 and SHA-256 both use six logical functions, where *each function operates on 32-bit words*, which are represented as x, y, and z. The result of each function is a new 32-bit word.

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{4.2}$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{4.3}$$

$$\sum\nolimits_0^{\{256\}}(x) = ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \tag{4.4}$$

$$\sum\nolimits_1^{\{256\}}(x) = ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \tag{4.5}$$

$$\sigma_0^{\{256\}}(x) = ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x) \tag{4.6}$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \tag{4.7}$$

SHA-256 64 32-bit constants:

4.2.2 SHA-224 and SHA-256 Constants

SHA-224 and SHA-256 use the same sequence of sixty-four constant 32-bit words, $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are (from left to right)

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

SHA-256 Hash Computation instructions:

6.2.2 SHA-256 Hash Computation

The SHA-256 hash computation uses functions and constants previously defined in Sec. 4.1.2 and Sec. 4.2.2, respectively. Addition (+) is performed modulo 2^{32}.

Each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed in order, using the following steps:

For i=1 to N:

{

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

2. Initialize the eight working variables, ***a***, ***b***, ***c***, ***d***, ***e***, ***f***, ***g***, and ***h***, with the (i-1)st hash value:

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ c &= H_2^{(i-1)} \\ d &= H_3^{(i-1)} \\ e &= H_4^{(i-1)} \\ f &= H_5^{(i-1)} \\ g &= H_6^{(i-1)} \\ h &= H_7^{(i-1)} \end{aligned}$$

3. For t=0 to 63:

{

$$\begin{aligned} T_1 &= h + \Sigma_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t \\ T_2 &= \Sigma_0^{\{256\}}(a) + Maj(a,b,c) \\ h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2 \end{aligned}$$

}

4. Compute the ith intermediate hash value $H^{(i)}$:

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} \\ H_4^{(i)} &= e + H_4^{(i-1)} \\ H_5^{(i)} &= f + H_5^{(i-1)} \\ H_6^{(i)} &= g + H_6^{(i-1)} \\ H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned}$$

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

Data processing:

```
import { readFileSync, writeFileSync, existsSync } from 'fs';
import { join } from 'path';

function imageToBase64(imagePath: string): string {
    if (!existsSync(imagePath)) throw new Error(`File ${imagePath} not found.`);
    return readFileSync(imagePath).toString('base64');
}

function saveBase64ToFile(base64String: string, outputPath: string): void {
    writeFileSync(outputPath, base64String);
}

const imagePath: string = join(__dirname, 'paste.jpg');
const outputPath: string = join(__dirname, 'paste.txt');

try {
    const base64Text: string = imageToBase64(imagePath);
    saveBase64ToFile(base64Text, outputPath);
} catch (error) {
    console.error(error.message);
}
```

Main code:

```
import { MerkleTree } from "merkletreejs";
const SHA256 = require("crypto-js/sha256");
import fs from "fs";

const charSim = 1 * Math.pow(10, -6); // 1 microsecond per character
const bs64_tides = fs.readFileSync("base64_tides.txt", "utf8");

function partitionString(string: string, numPartitions: number): string[] {
    if (numPartitions <= 0) return ["Invalid number of partitions"];
    const partitionSize = Math.ceil(string.length / numPartitions);
    const partitions: string[] = [];
    for (let i = 0; i < string.length; i += partitionSize) {
        partitions.push(string.substring(i, i + partitionSize));
    }
    if (partitions.length > numPartitions) {
        partitions[partitions.length - 2] += partitions.pop();
    }
    return partitions;
}

function mimicNetworkUnreliability(file: string, unreliability: number): string {
  return Math.random() < unreliability ? file.substr(0, file.length - Math.ceil(file.length * unreliability)) : file;
}

function requeryCorrupted(
    tree: MerkleTree,
    corruptedPartitions: string[],
    originalRoot: string,
    unreliability: number
): { root: string; time: number } {
    const leaves = tree.getLeaves();
    let localTime = 0;

    for (let i = 0; i < corruptedPartitions.length; i++) {
        const leaf = corruptedPartitions[i];
        const proof = tree.getProof(leaves[i]);
        const isValidLeaf = tree.verify(proof, leaf, originalRoot);

        if (!isValidLeaf) {
            let attempts = 0;
            while (attempts < 5) {
                localTime += (leaf.length + JSON.stringify(proof).length) * charSim;
                if (Math.random() >= unreliability) {
                    corruptedPartitions[i] = leaves[i].toString();
                    break;
                }
                attempts++;
            }
            // Add penalty for failed transmission, but make it less severe for smaller partitions
            if (attempts === 5) {
                localTime += leaf.length * charSim * 5 / Math.sqrt(corruptedPartitions.length);
            }
        }
    }

    // Adjust overhead for managing partitions
    localTime += corruptedPartitions.length * Math.log2(corruptedPartitions.length) * 0.00001;

    const finalTree = new MerkleTree(corruptedPartitions, SHA256);
    return { root: finalTree.getRoot().toString("hex"), time: localTime };
}

function testingPowers(unreliability: number): any[] {
    let results: any = [];
    for (let i = 1; i <= 12; i++) {
        const numPartitions = Math.pow(2, i);
        const partitions = partitionString(bs64_tides, numPartitions);
        const tree = new MerkleTree(partitions, SHA256);
        const corruptedPartitions = partitions.map(x => mimicNetworkUnreliability(x, unreliability));
        const root = tree.getRoot().toString("hex");

        const start = performance.now();
        const { root: finalRoot, time: localTime } = requeryCorrupted(tree, corruptedPartitions, root, unreliability);
        const end = performance.now();

        const baseTime = (bs64_tides.length * charSim) / Math.sqrt(numPartitions);
        const totalTime = (end - start) / 1000 + localTime + baseTime;
        results.push({ partitions: numPartitions, time: totalTime });
        console.log(`${totalTime.toFixed(3)}`);
    }
    return results;
}

console.log("Testing with 33% unreliability");
const results33 = testingPowers(0.01);
```

References

Becker, G. (2008, July 18). *Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis*. CiteSeerX. Retrieved August 14, 2024, from https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d7c3aa65bc5df32d94dcc8b29dceca240bdf8bef

Cachin, C., & Camenisch, J. (Eds.). (2004). *Advances in Cryptology – EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings*. Springer Berlin Heidelberg.

Elliott, B. J. (2000). *Cable Engineering for Local Area Networks*. Elsevier Science.

Katz, J., & Lindell, Y. (2021). *Introduction to Modern Cryptography*. CRC Press.

Merkle, R. C. (1987). *A digital signature based on a conventional encryption function*. People @EECS. Retrieved August 14, 2024, from https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf

NIST. (2015, August 4). *Federal Information*. NIST Technical Series Publications. Retrieved April 2, 2024, from https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

Penard, W., & van Werkhoven, T. (2008). *Chapter 1 On the Secure Hash Algorithm family*.

Tomescu, A., Crosby, S. A., & Wallach, D. S. (2020, December 22). *What is a Merkle Tree?* Decentralized Thoughts. Retrieved April 2, 2024, from https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/