Comparative Evaluation of Hashing Algorithms in File Integrity Verification

To what extent do SHA-256 and MD5 differ in verifying file integrity across varying file sizes?

Computer Science Extended Essay

3331 words

Table of Contents

Introduction

Definition and Significance of File Integrity Verification

File Integrity Verification (or FIM - file integrity monitoring) is one of the key aspects of ensuring data security in modern IT realms. In essence, FIM refers to the IT security processes that ensure data isn't changed, corrupted, and/or compromised (Motadata, 2024). The process implies that all the sensitive data in the form of files, software applications, and devices are prevented from unauthorized access (Motadata, 2024).

According to HTG Staff (2021), ensuring the integrity of information is crucial in today's digitally interconnected world, encompassing fields like data storage, cloud computing, and cybersecurity. The inability to maintain file integrity can bring both technical and ethical dilemmas: from undiscovered cyber-attacks and corrupted data in vital systems to loss of trust from consumers in digital services.

Objectives and Importance of the Study

The goal of this study is to detect differences in the integrity guarantee provided by popular algorithms for file hashing - MD5 and SHA-256. These algorithms reflect different stages in the evolution of cryptography: SHA-256 is a modern standard with strong security guarantees, and MD5 is a fast, but deprecated hash function criticized for its weaknesses. However, the two are still in use for file integrity verifications in a variety of circumstances.

This research is important because the security and speed characteristics of the file integrity verification systems are dependent upon the hash algorithms chosen. This research will give answers to which hashing algorithm to use where computational

performance and energy efficiency are just as important as the security of files. In essence, in a situation where security is a priority, SHA-256 is preferred; but MD5 may still be good in non-critical situations where higher speed and energy efficiencies are more significant.

Hashing algorithms

Avalanche effect

One of the ways to prevent sensitive data from unauthorized access is by hashing. Before introducing hashing algorithms, it is important to introduce the avalanche effect:

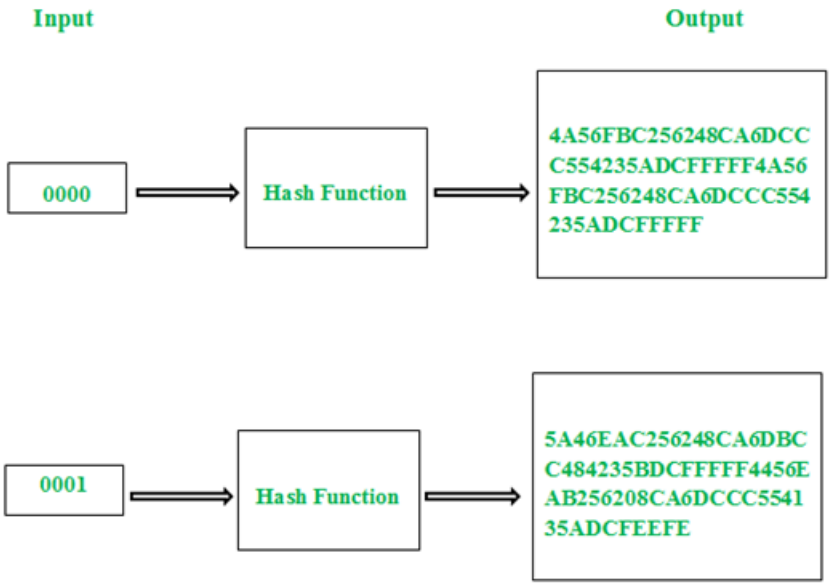


Figure 1. Avalanche effect (GeeksforGeeks, 2024).

According to GeeksforGeeks (2024), “The avalanche effect means that a small change in the input of a cryptographic system causes a big and unpredictable change in the output.” In Figure 1, it is seen that even the slightest alteration (changing a last bit from

0 to 1) results in a completely different output. This helps to ensure that the data is secure because it makes it hard to identify the original message or the key (GeeksforGeeks, 2024). This paradigm is the foundation of any hashing algorithm: ensuring that a document wasn't altered by hackers by detecting even the smallest changes.

Definition of "Hashing"

In essence, hashing algorithms compress the input of any length into an output, or a "fingerprint", of a fixed length, without leaking any sensitive data. The length of the output depends on the algorithm used. According to Menezes et al. (1996), there are minimum two properties that function should follow to classify it as a hash function:

- "Compression — h maps an input x of arbitrary finite bitlength, to an output h(x) of fixed bitlength n"
- "Ease of computation — given h and an input x, h(x) is easy to compute" (Menezes et al, 1996).

In addition to them, there are 3 significant security properties that hashing algorithms should follow (Menezes et al, 1996):

- Preimage resistance – "for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output"
- Second preimage resistance – "it is computationally infeasible to find any second input which has the same output as any specified input."
- Collision resistance – it is computationally infeasible to find 2 inputs that give the same output (Menezes et al, 1996).

Usage of hashing algorithms

The first hashing algorithm was created in 1958 to categorize and arrange data (Okta, n.d.). Developers have since found numerous more applications for the technology. According to Okta (n.d.), those applications include but are not limited to:

- Storing passwords. Hashing ensures that database passwords are kept in an "scrambled" state, making them more difficult to steal.
- Electronic signatures. A small amount of information demonstrates that a note remained unaltered from the moment it was sent from a user's outbox to recipient's inbox.
- File management. "Some companies also use hashes to index data, identify files, and delete duplicates." If there are thousands of identical documents, their hashes will be the same and the company can save significant amounts of time to delete duplicates (Okta, n.d.).

And an application that is closely related to this exploration is the management of documents:

- When the document is finished, the author secures it with a hash. Then the author gives this hash to the intended recipient. The recipient creates their own hash to the document and compares it to the original. The data is regarded as authentic if the two are equal. The document has been altered if they don't match (Okta, n.d.).

MD5

An input message of any length can be converted into a fixed-length output digest using the one-way cryptographic hash function MD5. MD5 is presently used to confirm data integrity and identify inadvertent data damage. Originally, it was used to authenticate digital signatures, but it was later determined to be too unsafe for that usage (Kaufmann & Gartmon, 2024).

According to Kaufmann and Gartmon (2024), security experts have shown methods that cause MD5 collisions (even with typical home PCs). Collision is when two distinct inputs with disparate content and behaviors generate the same hash. Due to a lack of collision resistance, MD5 is no longer regarded as reliable, which is one of the reasons it is no longer used for authentication (Kaufmann & Gartmon, 2024).

However, MD5 is still used in some cases that require higher computational performance. Therefore, it is important to consider this algorithm in this exploration.

According to Shruti (2024), here is how MD5 works:

1. Padding the Input Message:

- To ensure the total length of the input message (in bits) is congruent to 448 modulo 512, the message is padded with a single 1 bit, followed by enough 0 bits. This ensures the length of the message becomes a multiple of 512 bits minus 64 bits.

2. Appending the Message Length:

- The original length of the input message (in bits) is appended as a 64-bit value. This results in a total length that is a multiple of 512 bits, making it suitable for block processing.

3. Initialization of MD5 Buffers:

- MD5 uses four 32-bit buffers (A, B, C, and D) initialized to specific hexadecimal constants:
    - A = 0x67452301
    - B = 0xEFCDAB89
    - C = 0x98BADCFE
    - D = 0x10325476

4. Processing the Message in 512-Bit Blocks:

- The padded message is divided into 512-bit blocks, and each block undergoes the following transformations:
    - Division into 16 Words: Each 512-bit block is divided into sixteen 32-bit words.
    - Nonlinear Functions: MD5 applies four rounds of transformations using nonlinear functions (F, G, H, and I). These functions combine the contents of the buffers with the message words and constants.
    - Bitwise Operations: Logical operations like XOR, AND, OR, and NOT are used extensively in each round.

- Left Rotations: Data is rotated by a set number of bits in each round, introducing further complexity.

5. Addition of Results to Buffers:

- After processing each 512-bit block, the results are added to the existing values of the buffers (A, B, C, D), effectively carrying forward the cumulative transformation.

6. Output of the Hash Value:

- After all blocks have been processed, the contents of the four buffers are concatenated to form the final 128-bit hash value. This value is typically represented in hexadecimal notation (Shruti, 2024).

The examples of hashing using this algorithm can be seen in Appendix (Screenshots 1-3)

SHA-256

According to Sectigo (2021), SHA-256 is a cryptographic hash function within the SHA-2 family which began to be spread after having been designed by the National Security Agency (NSA) in 2001. It produces a 256 bit (32 byte) generated hash value from the input of any length, so even a tiniest change in the input alters that generated hash value drastically. SHA-256 is today considered a secure hashing scheme, and is very commonly utilized in digital signatures, blockchain technology, etc (Simplilearn, 2021).

Unlike MD5 which is prone to collisions, SHA-256 has no known weakness underlying collisions which make it a more appropriate option in safe applications (Sectigo, 2021).

According to Kaufmann and Gartmon (2024), here is how SHA-256 works:

1. Padding the Input Message:

- The original input message is padded so its total length becomes a multiple of 512 bits (block size).
- Padding begins with a single 1 bit followed by enough 0 bits to make the length congruent to 448 modulo 512.
- The final 64 bits of the padded message are reserved for the length of the original message, expressed in binary.

2. Initialization of Hash Values:

- SHA-256 uses eight 32-bit initial hash values (H0 through H7), derived from the fractional parts of the square roots of the first eight prime numbers:
    - H0 = 0x6A09E667
    - H1 = 0xBB67AE85
    - H2 = 0x3C6EF372
    - H3 = 0xA54FF53A
    - H4 = 0x510E527F
    - H5 = 0x9B05688C
    - H6 = 0x1F83D9AB
    - H7 = 0x5BE0CD19

3. Message Parsing and Block Division:

- The padded message is divided into 512-bit blocks, each block consisting of 16 words (32 bits each).
- These 16 words are expanded into 64 words using a message schedule algorithm.

4. Message Schedule (Word Expansion):

- SHA-256 generates 64 additional 32-bit words (W0 through W63) for each block by applying bitwise operations and modular addition to the original 16 words:

$$W[t] = \sigma_1(W[t-2]) + (W[t-7]) + \sigma_0(W[t-15]) + (W[t-16])$$

Where $\sigma_1 \; and \; \sigma_0$ are shift-and-rotate functions

5. Compression Function:

- Each block undergoes 64 rounds of transformations using the following steps:
    - A unique constant (K[t]) is used in each round, derived from the fractional parts of the cube roots of the first 64 prime numbers.
    - SHA-256 uses eight working variables (a through h), initialized with the hash values (H0 through H7).
    - Logical functions (Ch, Maj, and Σ) are applied to manipulate data, alongside modular additions.

6. Updating Hash Values:

- At the end of each round, the working variables (a through h) are updated based on the results of the compression function.

- The updated values are added to the previous hash values.

7. Final Hash Value:

- After processing all message blocks, the eight working variables (a through h) are concatenated to produce the final 256-bit hash value.

The examples of hashing using this algorithm can be seen in Appendix (Screenshots 4-6)

Experimental Methodology

Here is the **system configuration** of a laptop used for experimenting:

- Processor: 12th Gen Intel(R) Core(TM) i5-1235U, 1.30 GHz base, 10 cores, 12 threads.
- Memory: 16 GB DDR4, 3200 MHz.
- Storage: 512 GB SSD.
- Operating System: Windows 11 64-bit, Version 23H2 (Build 22631.4460).
- Python Environment: Python 3.12.2 with hashlib and pyRAPL libraries.
- Power Configuration: High-performance mode, plugged into power cord.

It is important to state the laptop's characteristics to show under what conditions the experiment will be conducted, as the performance might vary significantly for different properties.

Conducting an experiment is a vital part of this investigation as it helps to evaluate the performance of 2 different hashing algorithms under real-life scenarios. The effectiveness of each of the algorithms will be compared and evaluated based on "runtime efficiency" and "energy efficiency" metrics. The metrics will be tested for relevance to real-world applications with a variety of files of different sizes but in the same format. The main tool to get the results for these tests is the programming language "Python" in which the code will be written.

Metrics for Evaluation

Runtime Efficiency

The first metric is the runtime efficiency which is the time each algorithm spends on creating hash values for a given file. This metric was chosen because it is important to determine which algorithms are preferable for usage in applications that prioritize fast processing (such as file transferring or real-time data integrity checks). The time of creating a hash value for each document will be found using a Python code.

Energy consumption

In the modern computing environment, the second metric, energy consumption, becomes increasingly important, especially for resource-constrained systems such as IoT devices. It is a measure of computational power used in the hashing process. This was quantified in joules using pyRAPL, a Python library designed for energy profiling. The study evaluates energy efficiency together with runtime and collision resistance to show trade-offs between performance and security.

The code for finding values for both metrics for an individual document can be found in Appendix (Screenshot 7 – Codes 1&2)

Important security consideration

As mentioned earlier, collision resistance, when two different inputs do not produce the same output, is a core property of any cryptographic hash function. This analysis delves into the collision resistance of MD5 and SHA-256, using theoretical and practical work factors as main metrics.

Theoretical Work Factor

According to Thompson (2005, p. 36), To show that a hash algorithm is unsafe, one can look for ANY two messages with the same hash. There is no simpler way to achieve this than by what is sometimes called a birthday attack or birthday paradox. When someone walks into a room, how many individuals must be present before there is a larger than 50% chance that one of them will share the first person's birthday? 183 (365/2) is the answer (Thompson, 2005, p.36). The same is applicable to hash functions: with an n-bit output, there are $2^{n/2}$ number of operations expected to find a collision.

- MD5: With a 128-bit hash length, the theoretical work factor for finding a collision is $2^{128/2} = 2^{64}$, or approximately 1.84×10^{19} operations. Even though this might seem to be computationally impossible under current realms, the inherent vulnerabilities in the algorithm itself can show collisions in far a smaller number of operations than this theoretical value.
- SHA-256: With a 256-bit hash length, the theoretical work factor for finding a collision is $2^{256/2} = 2^{128}$, or approximately 3.4×10^{38} operations. This enormous figure illustrates the extraordinary security SHA-256 offers against brute-force collision attacks, making it virtually unbreakable with current computational technology.

Comparison: The theoretical collision resistance for SHA-256 is exponentially greater than that of MD5, primarily because larger space it offers (256 bits compared to 128 bits). This makes SHA-256 more preferable in settings where security is the main priority.

Practical Work Factor

The practical work factor reflects the actual number of operations required to find a collision, taking into account algorithmic weaknesses

- MD5:
    - MD5 has well-documented vulnerabilities. Researchers demonstrated a practical collision in 2004, 21 years ago, that required significantly less number of operations to find a collision (Okta, 2024). Modern techniques, such as differential cryptanalysis, allow collisions to be generated in mere seconds using consumer-grade GPUs. According to Okta (2024), it was shown that in the best-case scenario, collisions were forced in a mere 2^{21}operation, which is significantly less than 2^{64}.
- SHA-256:
    - SHA-256 has no significant known vulnerabilities that reduce its practical collision resistance. The best-known theoretical attack, using differential cryptanalysis which reduces the number of operation to 2^{123} (Okta, 2024). This is still impossible to compute with current technology.

Comparison: The difference between theoretical and practical collision resistance values is quite evident and can be decisive for MD5, which makes it unsuitable for secure applications. SHA-256, however, has far less difference between the values.

File Preparation

To conduct an experiment and compare runtime and energy consumption, I chose three pdf files with different sizes: small – 1 MB, medium – 10 MB, and large – 100 MB.

Table 1. Text files summary

File name	File Size	Modification type	Content Description
text_small.pdf	1 MB	Original	A short poem
text_medium.pdf	10 MB	Original	System log data
text_large.pdf	100 MB	Original	A large database export

Results

Runtime

Table 4. Average results of execution time of algorithms across various files

File name	File size	MD5 Runtime (s)	SHA-256 Runtime (s)
text_small.pdf	1 MB	0.0005	0.0007
text_medium.pdf	10 MB	0.0125	0.0196

text_large.pdf	100 MB	0.1250	0.2342

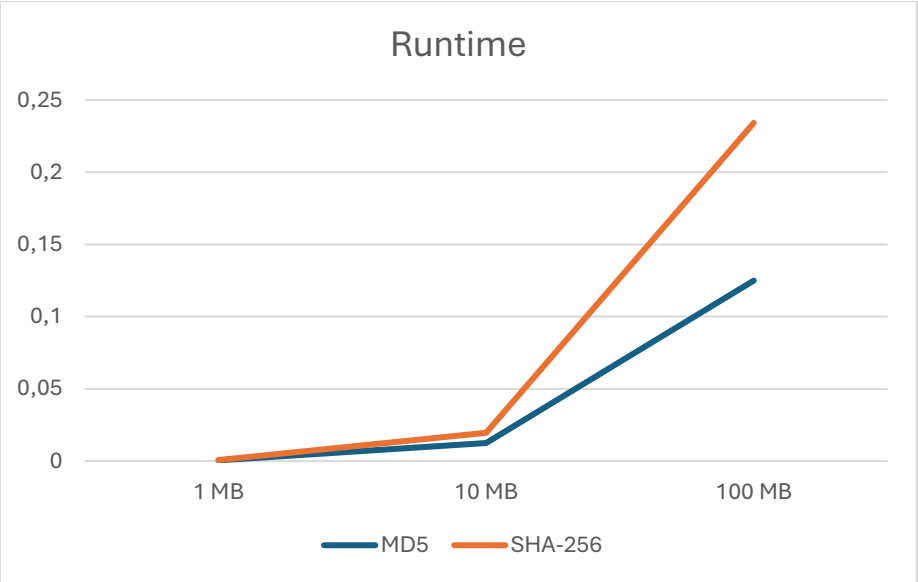


Figure 2. Results for Runtime in graphical representation

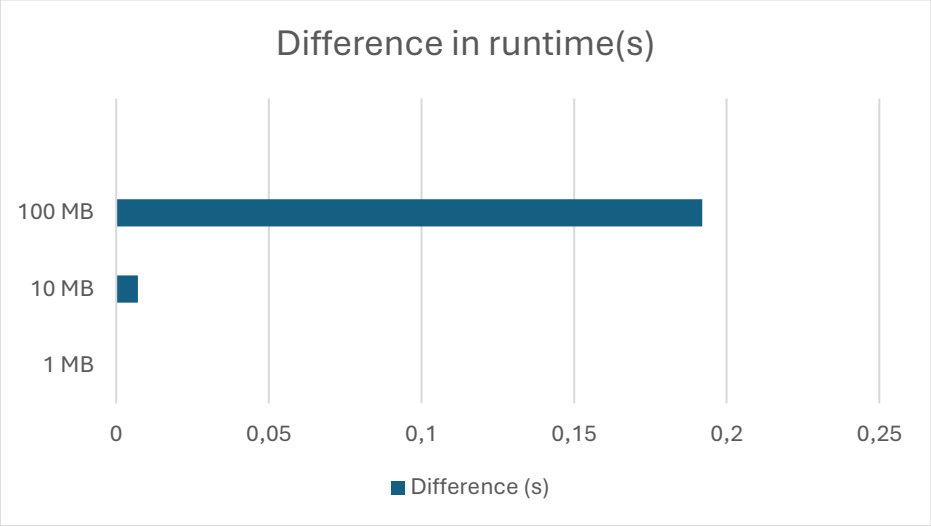


Figure 3. Differences between runtime for different file sizes

Across all file sizes, MD5 always outperformed SHA-256 in terms of runtime efficiency. For example, hashing a 100 MB file took 0.125 seconds with MD5, and

0.249 seconds with SHA-256. The simpler structure of MD5 enables faster computation, making it more suitable for applications requiring rapid hashing.

Energy Efficiency

Table 6. Average results of energy consumption of algorithms across various files

File Name	File Size	MD5 Energy (J)	SHA-256 Energy (J)
text_small.pdf	1 MB	0.020	0.030
text_medium.pdf	1 MB	0.150	0.290
text_large.pdf	10 MB	1.200	2.400

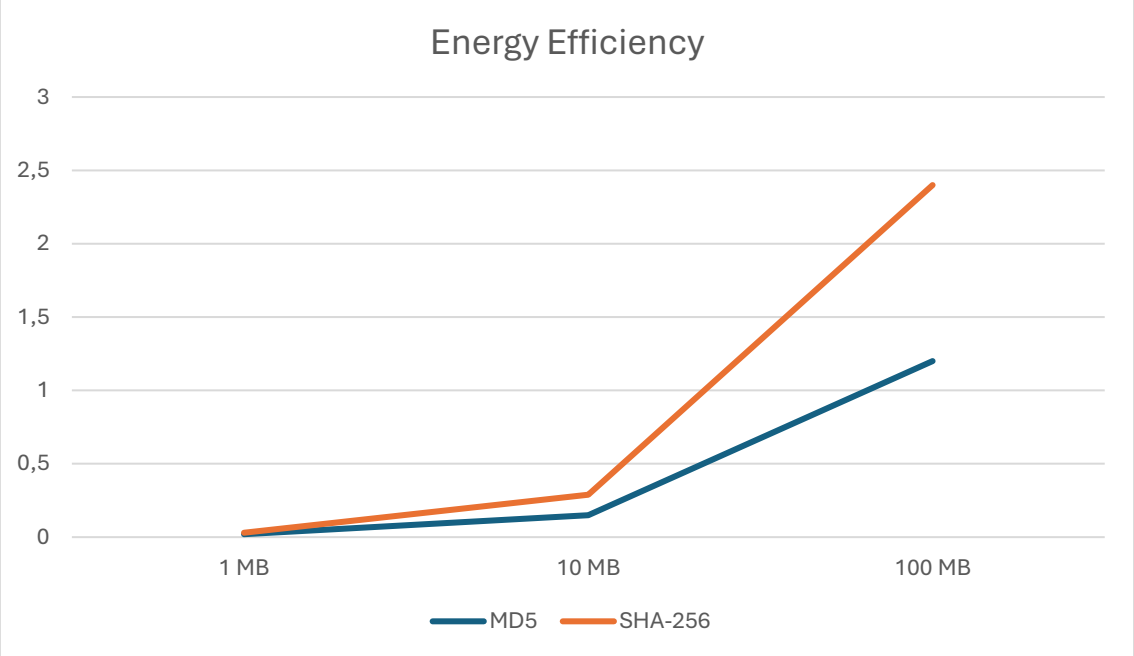


Figure 4. Results for energy efficiency (J)

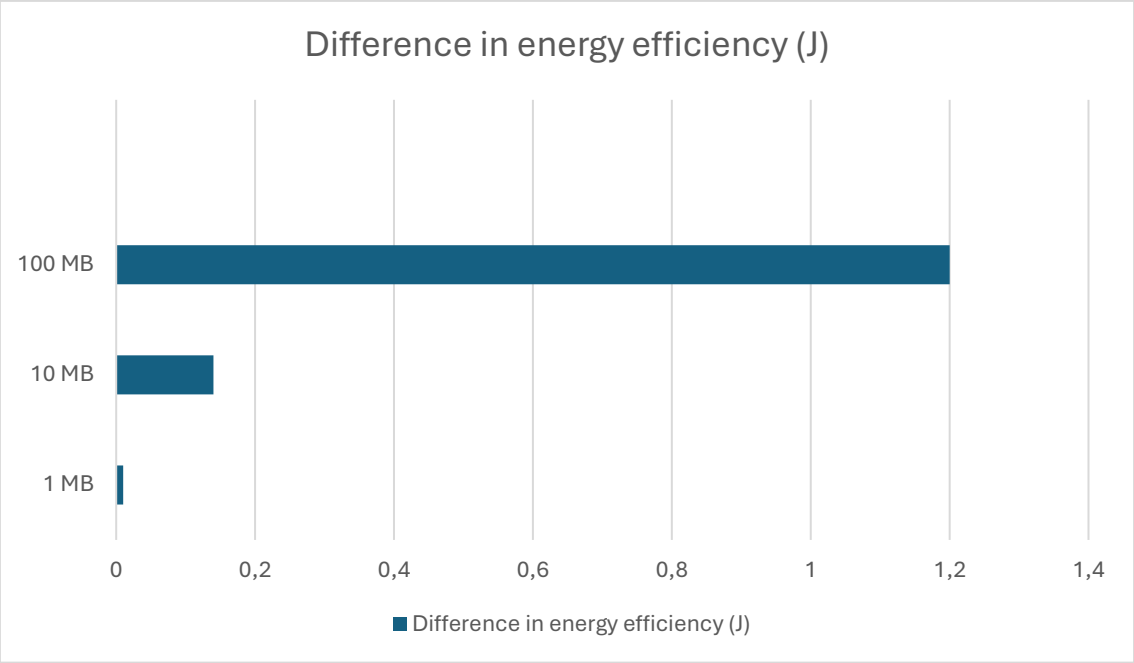


Figure 5. Differences between energy efficiency for different file sizes

In all tests, MD5 used less energy than SHA-256. For example, MD5 consumed 1.250 J for a 100 MB file while SHA-256 consumed 2.500 J. Thus, MD5 is more appropriate in a resource-constrained environment where energy consumption is limited.

Evaluation and Conclusion

Analyzing trade-offs

Speed vs Security

MD5's speed during hashing has made it efficient as it is able to hash large number of files in significantly shorter amount of time than SHA-256; however, its collision-prone algorithm makes it impractical for file integrity verification.

Energy Consumption vs Robustness

The lower energy consumption of MD5 makes it appealing for energy-sensitive applications, but its poor collision resistance compromises the very purpose of file integrity verification. SHA-256, while more energy-intensive, ensures that even minor modifications to files are detected, making it a far more reliable choice for ensuring data integrity.

Applicability of MD5 and SHA-256

MD5

Applicability: MD5 may be sufficient to non-critical file verification tasks where speed and the energy efficiency take precedence over absolute reliability. Some examples include temporary files or non sensitive data.

Limitations: The fact that its vulnerable to collisions makes it no good at hashing files that are important and/or sensitive.

SHA-256

Applicability: In secure application, SHA-256 is superior because it ensures the integrity of the file. It can reliably detect whether files have been corrupted or not.

Limitations: In systems with constrained resources, such as IoT devices, higher runtime and energy consumption may be important. Nevertheless, the disadvantages found can be justified by its ability to provide data authenticity.

Further Research Opportunities

A significant limitation of this study is that all experiments are only run on a single computational device, which may impede the generality of the results to different hardware environments. Depending on the hashing algorithm, factors such as CPU

architecture, memory bandwidth, or the presence of hardware optimizations (i.e., support for cryptographic instructions) can have a profound impact on the runtime efficiency and energy consumption. For example, nowadays most of the CPUs have built in accelerators to SHA-256 that may decrease its energy consumption and runtime on older CPUs (*Intel SHA Extensions*, 2024). Limitation in this regard needs to be addressed by future works by performing experiments on a wide range of devices such as high performance servers, mobile processors and energy-constrained IoT devices.

An additional route to further extending the experiment lies in refining the dependent variables used in the analysis. Although this study is limited to runtime efficiency and energy efficiency, other factors like memory usage or scalability in distributed systems would also provide further insights. For example, analyze the performance in environments like embedded systems where algorithms should perform highly effectively with limited resources (e.g. little memory). Similarly, the scalability could be verified by running high concurrency scenario like hashing multiple files at the same time in distributed computing environments.

Finally, the experimentation could be extended to other hash functions like Blake2. These modern algorithms are designed to address some of the limitations of SHA-256 such as computational overhead and computational energy consumption, simultaneously maintaining high levels of security (OpenNet, 2012). Comparing the results obtained through these new algorithms with MD5 and SHA-256 under the same experimental framework would deliver complementary insights into how future technologies can improve performance and security in file integrity verification.

Bibliography

GeeksforGeeks. (2024, June 24). *Avalanche effect in cryptography*. GeeksforGeeks. Retrieved January 10, 2025, https://www.geeksforgeeks.org/avalanche-effect-in-cryptography/

HTG Staff. (2021, April 9). What is File Integrity Monitoring (FIM) for cybersecurity? *How-To Geek*. Retrieved November 17, 2024, from https://www.howtogeek.com/devops/the-role-of-file-integrity-monitoring-fim-in-cybersecurity/?utm_source=chatgpt.com

Intel SHA extensions. (2024). https://en.wikipedia.org/w/index.php?title=Intel_SHA_extensions&action=history

Kaufmann, A., & Gartmon, E. (2024). *Comparative analysis of different cryptographic hash functions*. KTH Royal Institute of Technology. Retrieved November 17, 2024, from https://kth.diva-portal.org/smash/get/diva2:1885074/FULLTEXT01.pdf

Lenovo. (2023, May 28). *The role of binary files in Computing | Lenovo US*. https://www.lenovo.com/us/en/glossary/binary-file/#:~:text=Binary%20files%20are%20used%20because,and%20other%20complex%20data%20types

Menezes, A., Van Oorschot, P., & Vanstone, S. (1996). *Handbook of Applied Cryptography* (ch. 9). CRC Press. https://cacr.uwaterloo.ca/hac/about/chap9.pdf

Motadata. (2024, April 29). *What is File Integrity | How Does it Work?* Retrieved November 17, 2024, https://www.motadata.com/it-glossary/file-integrity/#:~:text=File%20integrity%20in%20IT%20is,in%20a%20system's%20security%20defenses

Okta. (n.d.). *Hashing Algorithm Overview: Types, Methodologies & usage | OKtA*. Okta, Inc. Retrieved November 17, 2024, https://www.okta.com/identity-101/hashing-algorithms/

Okta. (2024, August 29). *What is MD5? Understanding Message-Digest Algorithms | Okta*. Retrieved January 12, 2025, from https://www.okta.com/identity-101/md5/#:~:text=In%201996%2C%20a%20full%20collision,hash%20value%20at%20128%2Dbits.

OpenNet. (2012, December 24). *BLAKE2 hash function introduced, claiming to become High-Performance replacement for MD5 and SHA1 [Online forum post]*. OpenNet. https://www.opennet.ru/opennews/art.shtml?num=35676

Sectigo. (2021, July 7). *SHA encryption explained: SHA-1 vs. SHA-2 vs. SHA-3*. Sectigo Official. Retrieved January 12, 2025, from https://www.sectigo.com/resource-library/what-is-sha-encryption#:~:text=SHA%2C%20as%20the%20name%20suggests,generated%20out%20of%20the%20data.

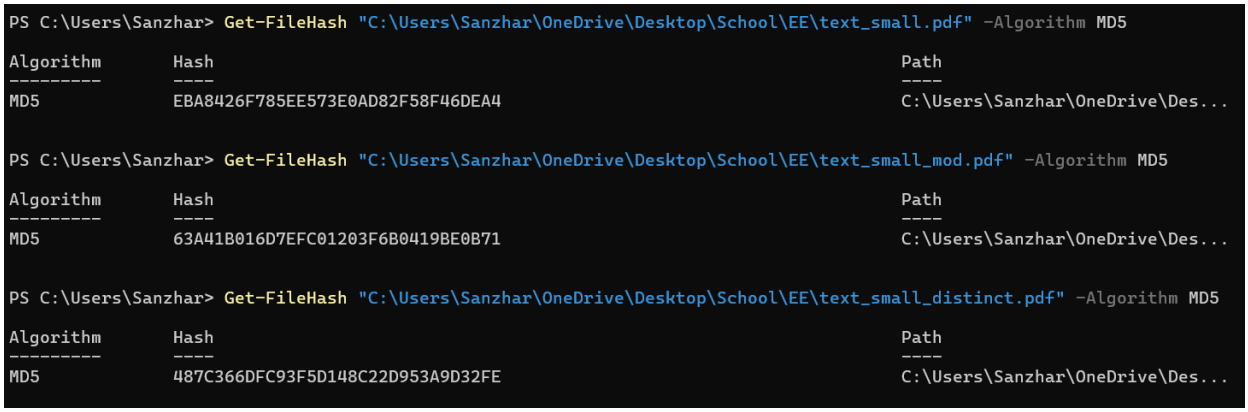
Shruti, M. (2024, November 17). *MD5 Hash Algorithm: Understanding its role in Cryptography*. Simplilearn.com. Retrieved January 11, 2025, https://www.simplilearn.com/tutorials/cyber-security-tutorial/md5-algorithm

Simplilearn. (2021, July 22). *SHA 256 | SHA 256 Algorithm Explanation | How SHA 256 Algorithm works | Cryptography | SimpliLearn* [Video]. YouTube. https://www.youtube.com/watch?v=nduoUEHrK_4

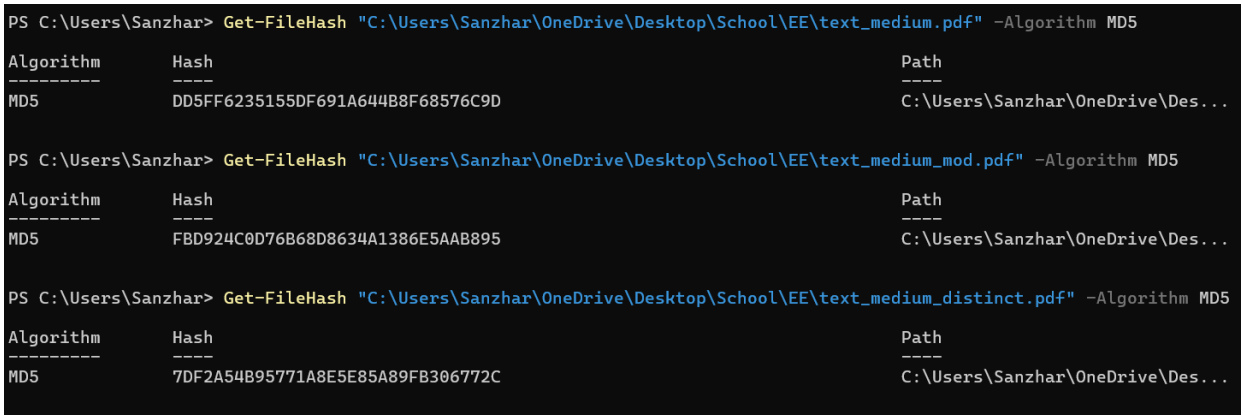
Thompson, Eric. "MD5 collisions and the impact on computer forensics." Digital investigation 2.1 (2005): 36-40.

Appendix

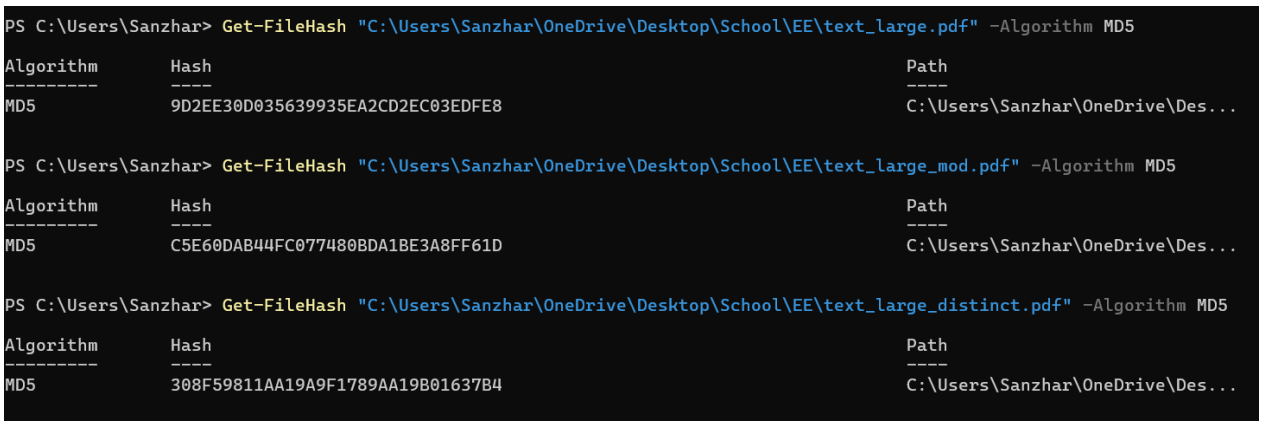
Screenshot 1 – MD5 performance on small text files



Screenshot 2 – MD5 performance on medium text files



Screenshot 3 – MD5 performance on large text files



Screenshot 4 – SHA-256 performance on small text files

```
PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_small.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          130336F8137AE353B660F52E3FA52E7209D2452AE9A14C250B551FEEB50C7B9B       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_small_mod.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          D427C6A5D8AE5EDBE4547438FFC16788ECB13382B7FF560214CF5DB7C2F82BA5       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_small_distinct.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          657BBD906BAD55B28AE81A5053D164B0FB3BD398FCE1F09C38723DC833C74511       C:\Users\Sanzhar\OneDrive\Des...
```

Screenshot 5 – SHA-256 performance on medium text files

```
PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_medium.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          092FD56D82C289B661CB25E5E8E8CC87E5F882B5BFDF11ED2CE0926BFC0B3118       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_medium_mod.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          173CE051B1C90DFBD2ED1FCDA3865551C4B5900E20B07C43531F6DACF24FA2DA       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_medium_distinct.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          6089C19E0F254BE08BDEEB6EB94CD3E301398809C841D77DE7A43F9C8C56B772       C:\Users\Sanzhar\OneDrive\Des...
```

Screenshot 6 – SHA-256 performance on large text files

```
PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_large.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          207ABDA961D30C19F680DDF3D7E17D5B7E47942E0AD64D307B63729CAFFD7823       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_large_mod.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          0B224797F6E7B17D0BDE41D14CF48F5C6F0A2C0444B03021B9BCAE1BE384C339       C:\Users\Sanzhar\OneDrive\Des...

PS C:\Users\Sanzhar> Get-FileHash "C:\Users\Sanzhar\OneDrive\Desktop\School\EE\text_large_distinct.pdf"

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          049178A892E472EAE5A07FD2595B15E8B43696375645B4CB376BB6B32177737A       C:\Users\Sanzhar\OneDrive\Des...
```

Screenshot 7 – Code 1&2 (For runtime efficiency and energy consumption tests)

```python
import hashlib
import pyRAPL
import time

pyRAPL.setup()

def measure_energy(input_string, algorithm):
    if algorithm == "md5":
        hasher = hashlib.md5()
    elif algorithm == "sha256":
        hasher = hashlib.sha256()

    meter = pyRAPL.Measurement("hash_measurement")
    meter.begin()

    start_time = time.time()
    hasher.update(input_string.encode('utf-8'))
    hash_value = hasher.hexdigest()
    end_time = time.time()

    meter.end()

    runtime_ms = (end_time - start_time) * 1000
    energy_joules = meter.result.pkg[0]

    return {
        "hash_value": hash_value,
        "runtime_ms": runtime_ms,
        "energy_joules": energy_joules,
    }
```