# Small Data Encryption for Database Storage Security and Speed

Isaac L. Kenyon
Iowa State University
Ames, Iowa
ilpenzol@iastate.edu

March 2025

***Abstract*—Modern database systems involve many different types and sizes of data being stored. Whether such data is in the range of terabytes or down to a single boolean bit, encryption ensures that it is safely stored at rest. However, the fact that some specific data is larger does not inherently mean that it is more valuable and, in turn, should be subject to more stringent encryption-at-rest standards. Small data can still involve very important and personal information. This information can range from, at times, generally public to private details, such as first and last name, address, and phone number, to Social Security Number, bank account number, and insurance ID. Industry and government alike have been implementing and writing laws regarding data encryption at rest. The IRS has stated in Publication 1075 on the Internal Revenue Code Section 6103 that the IRS must protect all the personal and financial information furnished to the agency against unauthorized use [7]. This research paper focuses on databases involving a large amount of small data, examines a current encryption algorithm in use, and proposes a new algorithm that focuses specifically on small data.**

## *I. Introduction*

Databases are the backbone of applications that store important information. End users often don't consider how such data is stored or whether their personal data is securely encrypted. A user's experience is heavily influenced by the speed of the application. However, ensuring strong encryption while maintaining performance is a technical balancing act. If encryption is too slow, it can degrade the user experience; if it's too weak, it leaves sensitive data vulnerable. This balance becomes especially critical for small but sensitive data, where both security and performance must be optimized without compromise. In *Figure 1*, a diagram illustrates the delicate tradeoff between security and speed. While this tradeoff is not a direct linear correlation, the figure highlights how and why algorithms are designed with different priorities.

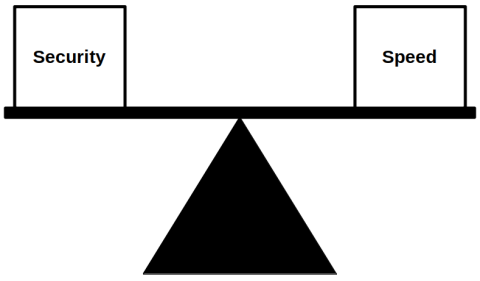



**Figure 1.** An example image showing the rudimentary trade-off between security and speed.

Currently, several algorithms are widely used. AES and ChaCha20 are the two discussed here. The cryptographic community warns that "non-standard or private" ciphers often have hidden flaws. Without public analysis, one cannot rely on their security. Such ciphers may be susceptible to simple attacks (e.g., distinguishing bias or chosen-plaintext) that AES and ChaCha20 resist. For example, if an algorithm uses a hash as a PRNG, its security hinges entirely on that hash's pseudorandomness—but a flawed use of a hash (no salt, low entropy) could leak information. Side-channel resistance is also unknown: a homebrew cipher likely doesn't take precautions against timing or memory-leakage attacks. In short, AES and ChaCha20 offer proven resistance to known attacks, whereas the proposed algorithm could be considered risky, as it has not stood the test of time and has not been subjected to extensive cryptanalysis.

## Advanced Encryption Standard

AES is a symmetric block cipher standardized by NIST in 2001 (FIPS PUB 197). It encrypts data in fixed-size blocks of 128 bits using keys of 128, 192, or 256 bits [10]. It is a worldwide standard and one of the most popular cryptographic primitives. The algorithm features non-linearity, where SubBytes adds confusion. AES has diffusion computation within ShiftRows and

MixColumns to spread bits across the state. It is designed to resist differential and linear cryptanalysis [5]. AES has survived numerous cryptanalytic efforts and, for that reason, has eventually made its way into mainstream use.

AES is used in nearly all major applications involving encryption due to its strength, efficiency, and wide acceptance. The AES algorithm is used in HTTPS for encrypting web traffic, VPN traffic, and within email applications. It is also widely used for data at rest, which is the use case in this paper. For example, AES is used in BitLocker, which is a Windows application [9]. Services like Google Cloud, AWS, and Dropbox also use AES to encrypt files and customer data [6]. Lastly, AES is implemented in a large number of crypto libraries such as OpenSSL, Libsodium, BouncyCastle, and Crypto++.

The AES algorithm is extremely secure and is the standard for most traditional use cases of data encryption. When encryption is needed for non-specific applications, AES is often a great starting point, though other approaches can outperform it in very specific implementations.

# ChaCha20 Algorithm

ChaCha20 is a variant of the ChaCha family of stream ciphers. It was designed to offer high performance in software, particularly where AES might be slow or vulnerable to side-channel attacks [3]. ChaCha20 differs from AES because it is a stream cipher, not a block cipher. The main implementation features a 256-bit key along with a 64-bit nonce. It operates on a 512-bit internal state made up of 4 constant words, 8 key words, 1 counter word, and 3 nonce words. After 20 rounds of mixing—the 20 in ChaCha20—it produces a keystream block. This keystream is XORed with the plaintext to encrypt it [8].

The ChaCha20 algorithm implementation is the choice of encryption when no hardware-accelerated AES is possible in the computer system. For this reason, ChaCha20 is very popular in the mobile phone and embedded system space due to hardware limitations. Google has integrated the ChaCha20 cipher into mobile and embedded devices through its development of the Adiantum encryption scheme. Adiantum was introduced to provide efficient full-disk encryption on low-powered devices, such as smartphones, smartwatches, and IoT devices, that lack hardware support for AES encryption [2].

This ChaCha20 algorithm is the main algorithm used as inspiration for this paper. Overall, ChaCha20 is quite a simple implementation, yet it has stood the test of time for security and has made its implementation very valuable in the specific fields stated above.

## *II. Proposal*

The point of this paper was to create a working novel approach for encryption that meets the requirements desired for small data. These requirements are listed below and were the basis of design decisions.

1. An encryption algorithm with a pure focus on small data, disregarding the efficiency or even security of large data inputs.
2. The encryption algorithm must be just as fast, if not faster, than current implementations used in applications that involve small or abnormally sized data types.
3. The implementation should be entirely software-based and initially must not rely on hardware optimizations.

An algorithm design is now going to be proposed that works fast on small data. The possible security implications will be discussed later on, though hopefully it provides just as good, if not better, security than the ChaCha20 algorithm. This algorithm also only needs a single hashed password, opting to exclude the input of a nonce (number-used-once) as seen in the ChaCha20 algorithm. The point of proposing a key or password that is smaller than the traditional 32-byte key + 12-byte nonce, as used in ChaCha20, is the storage savings seen in such a key system. The key system will also be discussed later on in terms of pure storage savings, not in any speed or implementation metrics.

Because the algorithm was designed with a focus on small data, the larger big-O runtime is seen as appropriate, as it still performs well on the targeted data. Later on, the runtime of larger data will be evaluated and possible modifications made to increase the data performance window size. The big-O runtime being polynomial is hoped to provide more movement of the data within the runtime of the encryption. Due to the nature of smaller data having fewer total combinations of arrangement, more computation is done to give the output a look of more randomization while still letting the algorithm function in reverse.

## *III. Design*

The main parts of this algorithm are the hash function, GetShift, GetDir, and the historical data input within the main for loop. They are further talked about below.

1. **Hash Function:** The hash function that was chosen for this specific implementation was the SHA-256 bit hash. This hash algorithm was chosen to provide diffusion among keys, safe storage of the users key, and the large keyspace.
2. **GetShift:** The GetShift function provides an integer value based off the char value, this is rangin from 0-15.
3. **GetDir:** The GetDir function provides a discrete way of random directon changes when shifting is taking place. It's function is to provide more pseudo-randomness to increase the number of brute-force attempts it would take.
4. **Historical Data Input:** The first implementation of this algorithm can only be performed in a sequential manner. The historical data input is used for the reason that when attempting to break an encryption, an incorrect guess for a specific byte would throw off the rest of the decryption attempt. The sequential-ness also provides an intuitive encrypt and decrypt sequence

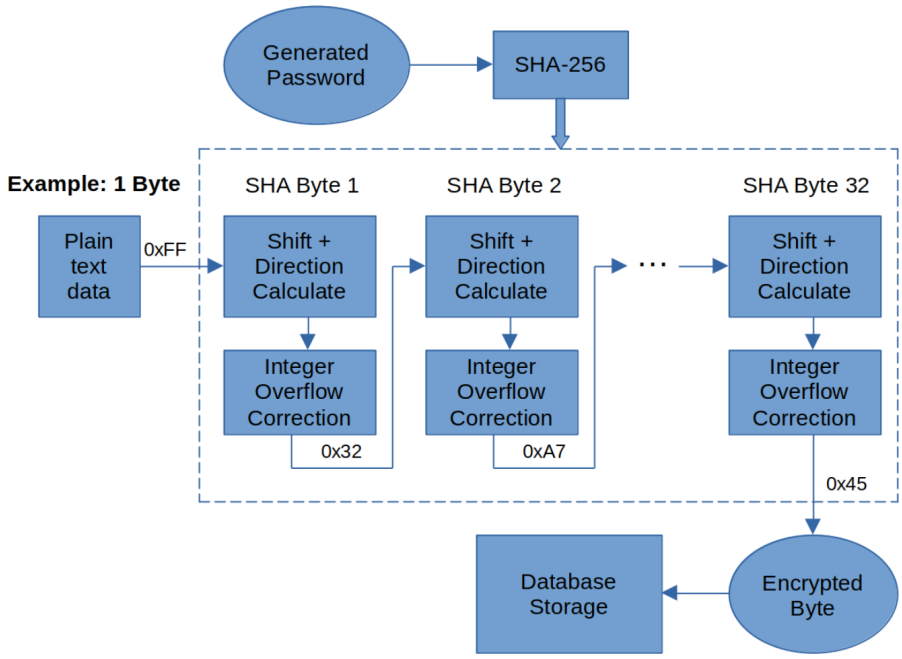



**Figure 2.** High level diagram of the proposed algorithm.

in terms of the implementation and attempts to minimize output similarities between keys that are also similar.

The sequential loop that is performed is done always 32 times for the 32 bytes that are present within the 256-bit SHA hash that is calculated from a key input. This correlates to 32 permutations similar to the 20 rounds seen in the ChaCha20 algorithm. Inside the main for loop modulus and modulus type operations are performed essentially "slicing" off the top of overflow values. This is once again done to add more encryption output space and pseudo-randomness, adding more combinations that would be needed to be checked.

The decryption of the algorithm is then done in reverse starting at the last byte of the hash algorithm and negation of the shift movements, while also starting at the end of the data byte array. For this reason the speed of the encrypt and decrypt is the theoretically the exact same as the same number of number computations is performed.

In *Figure 2*, a high-level design diagram shows how these parts interact with each other for the 32 bytes of the hash, and the input can be seen as an byte array. Also down in *Algorithm 1*, the pseudo-code can be seen showing how the functions are called and the inner functionality of the main for loop.

# Parallel Implementation

A parallel version of the algorithm was tested to see what the speed difference would be along with the security implications. In order for the parallel version to work, the chain effect would need to be removed so that operations on the entire array could be done in a parallel way.

This version did not end up being substantially faster, and the security was very poor as a simple glance at the output and input it could be seen that the bytes were shifted in a very elementary way.

After trying this version it was determined that the main reason that the initial implementation works is due to the chain-like effect. Down in *Algorithm 2*, the implementation can be seen. It would be interesting to see what other parallel implementations could be used yet still keep the chain effect to a point.

# Single Implementation

A single implementation of the algorithm was also tested to see the speed of a single permutation and look at the fundamental security of the one single pass. This gained insights into how each permutation adds layers to the security and showed how at the very basic implementation of the algorithm, it is not very secure. This version is also quicker in a linear fashion, essentially dividing the time by the number of permutations taken from the original 32 passes.

**Algorithm 1. Scramble Byte Array Using Hash**

```
1: procedure SCRAMBLE(array, hash)
2:     for i = 0 to length(hash) do
3:         c ← hash[i]
4:         shift ← getShift(c)
5:         dir ← getDir(shift)
6:         shift ← shift × dir
7:         for j = 0 to length(array) − 1 do
8:             if j ≠ 0 then
9:                 amount ← array[j] + shift + array[j − 1]
10:            else
11:                amount ← array[j] + shift + array[j]
12:            end if
13:            while amount > MAX do
14:                amount ← MIN − 1 + (amount − MAX)
15:            end while
16:            while amount < MIN do
17:                amount ← MAX + 1 − (MIN − amount)
18:            end while
19:            array[j] ← amount
20:        end for
21:    end for
22:    return array
23: end procedure
```

**Algorithm 2. Compressed Scramble Using Hash**

```
1: procedure COMPRESSEDSCRAMBLE(array, hash)
2:     startTime ← current time (nanoseconds)
3:     shift ← compressHash(hash)
4:     dir ← getDir(shift)
5:     shift ← shift × dir
6:     for j = 0 to length(array) − 1 do
7:         change ← 0
8:         if j ≠ 0 then
9:             amount ← array[j] + shift + array[j − 1]
10:        else
11:            amount ← array[j] + shift + array[j]
12:        end if
13:        while amount > MAX do
14:            edit ← amount − MAX
15:            amount ← MIN − 1 + edit
16:        end while
17:        while amount < MIN do
18:            test ← MIN − amount
19:            amount ← MAX + 1 − test
20:        end while
21:        array[j] ← amount
22:        newByte ← change
23:    end for
24:    return array
25: end procedure
```

**Algorithm 3. Parallel Scramble Using Hash**

```
1:  procedure ParallelScramble(array, hash)
2:      Initialize thread pool
3:      startTime ← current time
4:      for i = 0 to length(hash) − 63 do
5:          c ← hash[i]
6:          shift ← getShift(c)
7:          dir ← getDir(shift)
8:          updated_shift ← shift × dir
9:          range ← MAX − MIN + 1 − updated_shift
10:         parallel for all j in 0 to length(array) − 1:
11:             amount ← array[j] + updated_shift
12:         if amount > MAX then
13:                 amount ← MIN + ((amount − MIN) mod range)
14:         else if amount < MIN then
15:                 amount ← MAX − ((MIN − amount − 1) mod range)
16:         end if
17:             array[j] ← amount
18:     end for
19:     endTime ← current time
20:     Shutdown thread pool
21:     return array
22: end procedure
```

## *IV. Results*

Extensive testing across varying data sizes revealed the proposed algorithm and its variants shine with smaller datasets. In these instances, it delivers rapid execution and low latency, crucial for applications like embedded systems or real-time processing with tight deadlines. However, performance noticeably declines as data size grows, a common trait reflecting increasing computational complexity. While this degradation at larger scales exists, this paper primarily focuses on the algorithm's strong performance with small datasets, its intended domain. This ability to quickly process small data is a key strength, making it ideal for resource-constrained or time-sensitive scenarios. Although scalability and performance on larger datasets warrant future investigation, they fall outside this study's scope. In essence, the algorithm's is useful with small datasets—a significant and an advantage for many applications—takes precedence over its behavior with larger ones within the context of this research.

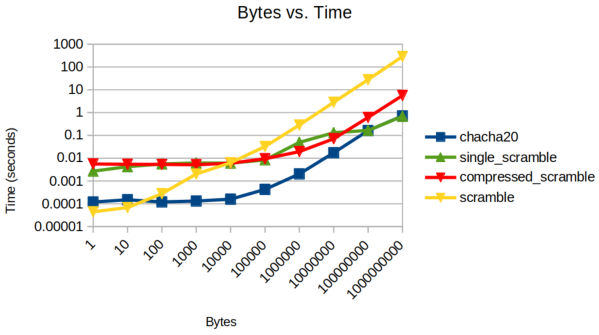



**Figure 3.** A graph showing the Bytes vs. Time of the different proposed algorithms and ChaCha20.

# Algorithm Upsides

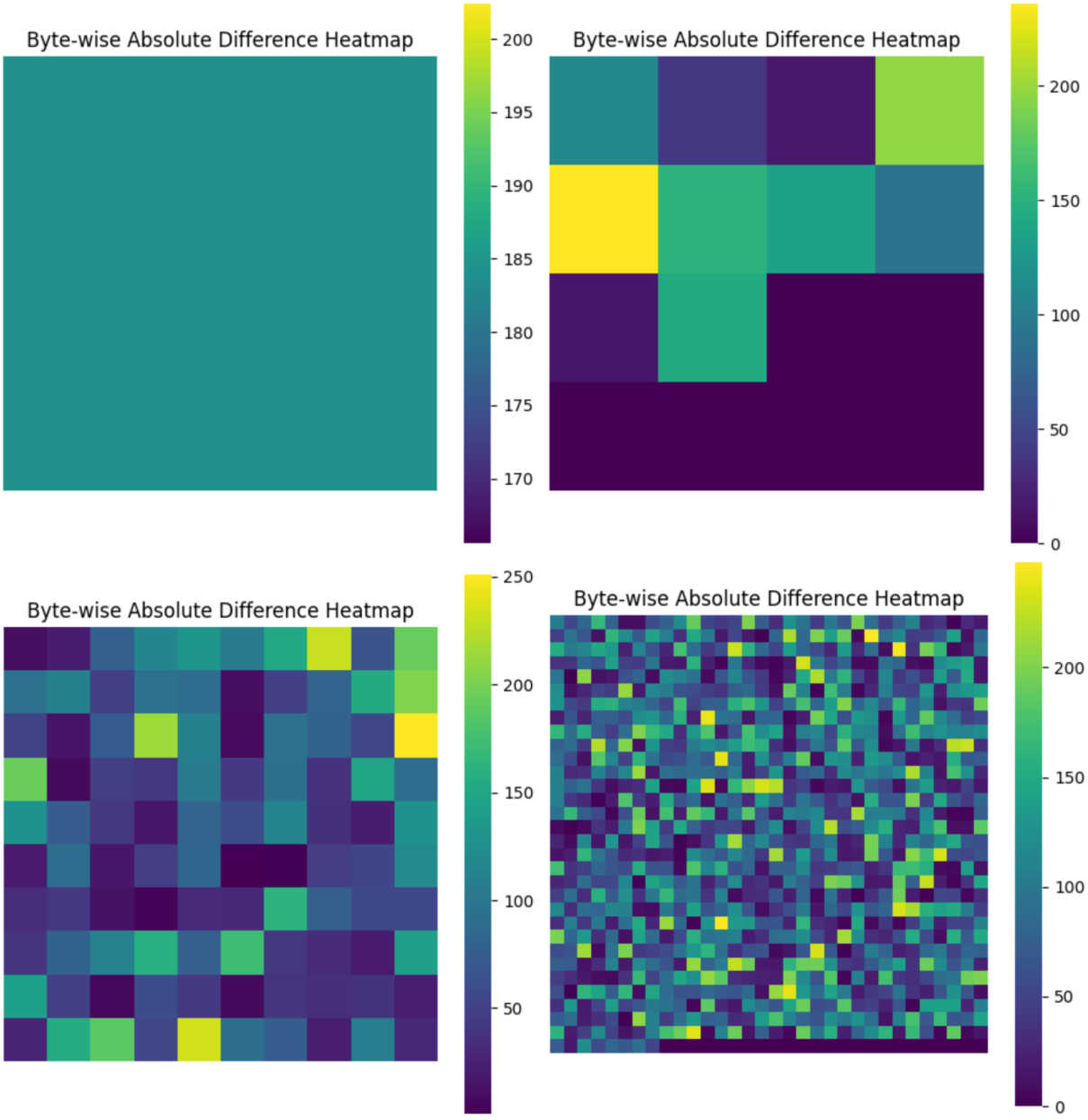
The main possible upsides of this algorithm is the simplicity to implement and understand, along with being very lightweight. It is deterministic and easily reservable when the hash is known, it also features pseudo-diffusion thanks to the array[j-1] feature that creates a chain like effect.

**Simplicity and Understandability:** The algorithm has a very simplistic basis and could be a great starting point for future changes. The simplistic implementation can also make it easier to detect security vulnerabilities as the algorithm won't be making redundant operations that could be skipped right passed when attempting to break an encryption.

**Lightweight:** The algorithm is light weight and can be implemented in software and essentially on any system. It could run very well on slower systems within the embedded world and mobile devices. This was one of the targets for the design.

**Deterministic:** Determinism when the key and or hash is know is very important if decryption is to be possible. It is very clear that the decryption essentially just involves computation in reverse.

**Pseudo-diffusion:** Since the algorithm features a chain like effect seemingly linear changes are pseudo-randomized. This adds a layer of complexity to the security and stops the changes from just being a single linear change.




**Figures 4-7** Byte-wise absolute heatmaps of same password input comparison of randomized input data vs output with 1 byte, 10 bytes, 100 bytes, and 1 kilobyte file sizes respectively.

# Algorithm Downsides

Discussing possible downsides that were noticed during the writing and testing of the algorithm is very important in the process of determining if such algorithm is suitable for purposes in industry, not just academic. This algorithm has shown multiple problems. The first problem is going to be called the Same Key Similarity. The second problem that has been observed is called differential threats.

**Same Key Similarity:** When assuming that a large amount of small data is going to be encrypted with the same exact key, similarity problems arise. This is due to the way the main implementation of the algorithm requires past manipulated data as input into the current byte being encrypted. When these input keys are the same, the individual bytes as the algorithm runs will be the same in all cases, up until the first input character is different. This problem would generally not matter much for general encryption algorithms. However, because of the nature of small data, similar or exactly the same values could be present.

**Similar Key Problem:** An encryption algorithm that has real-world use cases should be susceptible to slight changes to its secret key. Attackers could possibly break the encryption algorithm using a similar key, so any small change in the key used in the decryption step should not allow the attacker to do so [1].

**Differential Threats:** This method of attack is used to evaluate the differences in the encryption output due to slight variations of the input [4]. This problem is similar to the Same Key Similarity, but this time the keys are different and the input data is very similar.

**Cryptographic Randomness:** Having a deterministic output is also a downside, especially since a nonce is not used for this algorithms implementation. The transformations are linear and predictable, along with the hash key not being processed as it is in secure implementations like ChaCha20 and AES.

## *V. Proposed System*

This proposed system was designed under the principle that security can be improved by minimizing the time sensitive data spends in transit, at rest, or in active use. By reducing exposure in these three critical states, the system mitigates the risk of interception, leakage, or unauthorized access. Given that the primary use case involves small data payloads, the system anticipates a high frequency of lightweight HTTP or similar protocol requests, making performance under rapid, repeated access a top priority.

To address this, it is proposed that the algorithm introduced in this paper be integrated into the system's data handling pipeline. The algorithm's simplicity and speed make it well-suited for environments where low latency is critical and where a large number of concurrent read/write operations must be processed efficiently. Unlike traditional cryptographic ciphers that often require block padding (e.g., AES in CBC mode), this algorithm operates on raw byte streams without padding, thereby reducing storage overhead and simplifying data alignment.

In addition, because the algorithm was designed to function without a nonce—unlike standard stream ciphers such as ChaCha20—it eliminates the need to generate, store, or transmit nonces alongside encrypted data. This further reduces metadata overhead, simplifies implementation, and minimizes the footprint of secure key storage. While nonce-free designs can introduce risks if not handled carefully, this trade-off is acceptable in controlled environments where the same key is not reused across identical inputs.

Collectively, these design choices contribute to a more efficient and practical security model for constrained or high-throughput systems that prioritize performance, simplicity, and minimal resource consumption, especially in embedded, mobile, or distributed edge environments.

## *VI. Conclusion*

In conclusion this algorithm is going to need it's security heavily tested. No algorithm can be quickly adopted without showing that it is not going to be a vulnerability within a system. As of right now this algorithm does not stand up to ChaCha20 and AES in terms of security. It has many pitfalls listed below.

After analyzing the algorithm it would not be recommended to use it in it's current state for any systems that require security. After performing tests on the algorithm it is able to obfuscate data but it does not go the length to be considered secure encryption.

1. **Deterministic Output:** Deterministic encryption suffers from a critical flaw: it betrays patterns. The consistent encryption of identical data allows attackers to detect repetitions, thereby leaking information about the underlying plain text. In contrast, secure encryption thwarts this by ensuring that each encryption of the same input yields a unique cipher text, typically through the incorporation of random elements like a nonce or Initialization Vector (IV).
2. **Hash as a Key:** Hash functions aren't designed to produce secure symmetric keys by themselves. Proper keys should be uniformly random, derived with secure functions like PBKDF2, scrypt, or Argon2, and be of appropriate length for the encryption algorithm. Using the raw hash can make brute force easier and fail to provide entropy.
3. **Linear and Predictable Transformation:** This algorithm is reversible and often algebraically simple. Also because of the linearity, patterns in plain text often survive or show up in cipher text. For that reason, attackers can deduce the transformation through statistical or differential attacks.

## *VII. Further Research*

The security of the proposed algorithm was not extensively studied and much room is open to subject the algorithm to common security checks. The downsides section does give a look at possible problems noticed as the algorithm was being designed and written. As there are many different aspects of security, below are listed some of the main aspects of determining an encryption algorithms attack resilience.

1. **Mathematical Foundations:** When breaking down the algorithm what is the mathematical model at the lowest level. Is there a formal proof that can ensure breaking the encryption is just as hard as the mathematical problem used for the encryption.
2. **Classical and Modern Cryptanalysis:** Brute-force, linear and differential, side-channel attacks are common approaches one might take for attacking an algorithm. This proposal has not been checked for a wide variety of attacks.
3. **Algorithm Structure:** How well does the algorithm diffuse the data among itself? When visualizing the output of the data and comparing to the input are there common indications that pop up to give an attacker insights to the key and or input?
4. **Peer Review and Standardization:** Having many eyes on an encryption implementation is best for determining the security as researchers specializing in specific attacks can test this algorithms resistance.

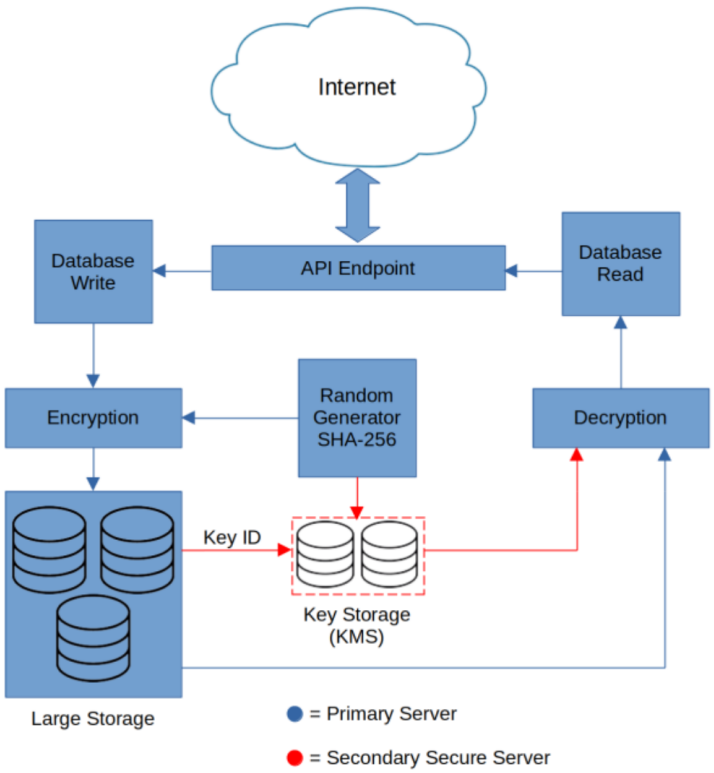



**Figure 8.** A theoretical system that could be used when designing the proposed algorithm.

# References


[1] Abdullah Alghamdi and Paul C. van Oorschot. Protecting data confidentiality against strong physical attacks. In *2021 IEEE 44th Symposium on Security and Privacy (SP)*, pages 1148–1165. IEEE, 2021.

[2] A. Author, B. Author, and C. Author. Title of the paper. *Transactions on Symmetric Cryptology (ToSC)*, Volume Number(Issue Number):Start Page–End Page, Year. Accessed: 2025-05-10.

[3] Daniel J. Bernstein. Chacha, a variant of salsa20. `https://cr.yp.to/chacha/chacha-20080128.pdf`, 2008. Accessed: 2025-05-09.

[4] Yilei Ding, Rahul Thakur, and Kassem Fawaz. Confidentiality isn't enough: Privacy attacks exploiting unencrypted auxiliary data in encrypted traffic. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1207–1224. IEEE, 2023.

[5] Christophe Giraud. Dfa on aes. Technical Report 2003/010, IACR ePrint Archive, 2003. Available at `https://eprint.iacr.org/2003/010.pdf`.

[6] Google Cloud. Encryption at rest. `https://cloud.google.com/security/encryption-at-rest`, 2024. Accessed: 2025-05-09.

[7] Internal Revenue Service. Encryption requirements of publication 1075. `https://www.irs.gov/privacy-disclosure/encryption-requirements-of-publication-1075`, 2021. Accessed: 2025-05-07.

[8] Adam Langley, Yoav Nir, and Seth Schoen. Chacha20 and poly1305 for ietf protocols. `https://datatracker.ietf.org/doc/html/rfc8439`, 2018. RFC 8439, Accessed: 2025-05-09.

[9] Microsoft. Bitlocker overview. `https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/`, 2024. Accessed: 2025-05-09.

[10] National Institute of Standards and Technology. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001. Available at `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`.