Proceedings of the 2nd International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/109/2024.17894

Network Security Applications in the Internet of Things

Juncheng Zhou

University of Utah, Salt Lake City, 84112, United States

zjc1403109@163.com

Abstract. In the realm of Internet of Things (IoT) network security, encryption algorithms serve as one of the core means to protect data and device security. Given the extensive connectivity of IoT devices to the Internet, ensuring security becomes a paramount challenge. Encryption algorithms effectively guarantee confidentiality, integrity, and authentication during data transmission while thwarting malicious attackers from unauthorized access or tampering. Therefore, this paper proposes employing hash storage for encrypting IoT data to enhance its protection. The subsequent sections will demonstrate how the data is stored and encrypted through code implementation. The method adopted in this research involves writing code and testing code to assess its reliability in terms of security measures. This investigation reveals that utilizing hash storage encryption of data provides a robust means for managing data securely within an IoT framework; moreover, longer encrypted data lengths contribute to enhanced security levels with the adoption of the SHA-256 algorithm.

Keywords: Hash algorithm, SHA-256, cybersecurity, protocol, anti-premapping.

1. Introduction

The rapid development of the Internet of Things (IoT) has connected a wide range of devices, from sensors and smart homes to industrial equipment, enabling real-time data exchange across various sectors like smart cities, transportation, and healthcare. However, this connectivity brings significant cybersecurity challenges. The heterogeneity of IoT devices, which use different operating systems and protocols, complicates the establishment of unified security measures. Many IoT devices are resource-constrained, lacking the computing power and storage needed for traditional encryption and authentication methods. Additionally, the large-scale nature of IoT networks increases the potential of attacking the surface, making them more vulnerable to cyberattacks like Distributed Denial of Service (DDoS) [1]. And an attacker can affect the entire network by hacking one device [2]. Data privacy is another major concern, especially with the increasing collection of sensitive user data. IoT systems must comply with stringent data protection regulations like GDPR, but ensuring security and privacy during data transmission and storage remains a challenge. Existing research in IoT security has several shortcomings. These include the lack of uniform security standards across devices, difficulty in managing security for resource-limited devices, and challenges in ensuring security in dynamic and large-scale networks. Additionally, privacy protection measures and supply chain security need further attention. While blockchain and smart contracts show promise for IoT security, integrating them into resource-constrained environments remains difficult [3]. In summary, despite advancements, IoT cybersecurity faces many unresolved issues. The heterogeneity of devices, resource limitations, and

DOI: 10.54254/2755-2721/109/2024.17894

challenges in privacy protection and large-scale security need more comprehensive research. Addressing these gaps is essential for building more robust and efficient IoT security solutions in the future.

2. Literature review

The resources of IoT devices are limited, and traditional encryption algorithms are difficult to operate efficiently. Although lightweight encryption algorithms (such as lightweight AES and ECC) exist, their performance and security in practical applications still need to be further optimized and verified. Second, in large-scale IoT networks, the generation, management, and distribution of keys is a huge challenge. The traditional centralized key management is no longer applicable. How to realize decentralized and distributed key management is an important research direction. At the same time, device authentication is the basis to ensure the security of the Internet of Things. The existing research mainly focuses on the software level, and the anti-tamper measures of the device itself are not perfect. How to ensure the authenticity and security of the device identity is a problem to be solved. Next, IoT devices collect large amounts of sensitive data, and while encryption protects privacy, it also affects the performance of the device. How to realize privacy protection on the premise of maintaining efficient operation is a blank in current research. As IoT devices frequently join or leave the network, how to maintain communication security in a dynamic network environment is a challenge. Most of the existing studies focus on static networks, and the security of dynamic environments is insufficient. Now, blockchain is also gradually becoming a hot spot in encryption technology, and while blockchain has the potential to be applied to the Internet of Things in terms of decentralization and security, especially in key management and authentication, how to apply it in resource-constrained devices remains to be studied. In terms of supply chain, the supply chain of IoT devices involves multiple links, and supply chain attacks may threaten device security. The current research mainly focuses on the equipment itself, while the security protection scheme of the entire supply chain is still insufficient. In summary, there are still research gaps in IoT encryption technology in lightweight, key management, dynamic network, privacy protection, large-scale attack protection, and other aspects. Improving research in these areas will improve the overall security of IoT and drive its wider application [4].

3. Methods

The algorithm for implementing encryption this time is SHA-256 Hash encryption storage algorithm. The hash function converts the input data (called a message) into a fixed-length hash value (called a summary). Hash values are usually expressed in hexadecimal form, and SHA-256 generates a hash value of 256 bits (32 bytes) in length. Security: Collision resistance: Two different inputs that compute a SHA-256 hash will not produce the same hash. Even with the rare possibility of a hash collision, it is hard to find two different inputs that generate the same hash value. Anti-premapping: For a given hash value, it is almost impossible to calculate the original input data. Anti-second premapping: For a given input data and hash value, it is very difficult to find another input data that is different and produces the same hash value. Structure: The SHA-256 algorithm divides the input data into 512-bit blocks that are processed in 64 rounds, each using a constant and the previous hash value. After 64 rounds of processing, the resulting 256-bit hash is the final output [5].

4. Application of SHA-256

4.1. Preparation process

Data integrity: verify that data has not been tampered with during transmission or storage. It is often used to generate and verify the hash value of a file when downloading it. Digital signature: in a digital signature, the hash value of the data is encrypted to ensure data integrity and authentication. SHA-256 is used in digital signature algorithms such as RSA and ECDSA [6]. Password storage: used to securely store user passwords. By taking a SHA-256 hash of the password and storing the hash value, the actual password is not directly compromised, even if the database is compromised. Blockchain technology: The hash value of each block in the blockchain is calculated based on the hash value of the previous

DOI: 10.54254/2755-2721/109/2024.17894

block. SHA-256 is the core hash function in Bitcoin and many other blockchain systems. Key generation: The process used to generate a key, ensuring that the key has sufficient entropy and randomness.

4.2. Concrete code implementation

The following shows hash algorithm code 1:

```
    import hashlib
# Step 1: Define the data that you want to hash
data = "Hello, this is a sample data to hash."
    # Make sure 'data' is defined before use
# Step 2: Convert the data to bytes
data_bytes = data.encode('utf-8')
# Step 3: Generate the hash using SHA-256
hash_object = hashlib.sha256(data_bytes)
    hash_hex = hash_object.hexdigest()
    # This is the hash value in hexadecimal format
# Step 4: Store the hash value in a file
filename = "hash_storage.txt" with open(filename, 'a') as f:
    # Writing the data and its hash value to the file    f.write(f"data: {data}\nhash: {hash_hex}\n\n")
# Output message to confirm the process is successful    print(f"Data and its hash value are stored in
{filename}")
# Suppose we want to store data and hash values in a file    filename = "hash_storage.txt"
# Open the file and store the raw data and the corresponding hash value
with open(filename, 'a') as f:
      f.write(f"data: {data}\nhash: {hash_hex}\n\n")
print(f"Data and hash values have been stored to {filename}")
    def verify_hash(stored_hash, new_data):
      # Convert the new data to a byte stream
      new_data_bytes = new_data.encode('utf-8')
      # Regenerates the hash
      new_hash_object = hashlib.sha256(new_data_bytes)
      new_hash_hex = new_hash_object.hexdigest()
      # Compare two hash values
      return stored_hash == new_hash_hex
# Let's say we have some new data to verify
new_data = "Hello, this is a sample data to hash."    is_valid = verify_hash(hash_hex, new_data)
if is_valid:    print("The data is verified and not tampered with")
else:    print("Data validation failed and may have been tampered with")
def store_hash(data, filename="hash_storage.txt"):
      # Generate hash
      data_bytes = data.encode('utf-8')
      hash_object = hashlib.sha256(data_bytes)
      hash_hex = hash_object.hexdigest()
      # Store to file
      with open(filename, 'a') as f:
            f.write(f"data: {data}\nhash: {hash_hex}\n\n")
      print(f"Data and hash values have been stored to {filename}")
    def load_and_verify(data, filename="hash_storage.txt"):
      # Read the hash value in the file
      with open(filename, 'r') as f:
            content = f.read()
            if f"data: {data}" in content:      print("data exist,In verification...")
```

DOI: 10.54254/2755-2721/109/2024.17894

```
            # Find the corresponding hash value in the file
            start = content.index(f"data: {data}") + len(f"data: {data}")
            end = content.index("\n", start)
            stored_hash = content[start:end].strip()
            # Validation hash
            return verify_hash(stored_hash, data)
        else: print("Data does not exist")
            return False
# Stores data and its hash value
store_hash("Sample data to hash.")
# Verify that data has been tampered with
result = load_and_verify("Sample data to hash.")
print("Data validation result:", result)
```

4.3. Explanation

The hashlib library provides access to different hash functions, including SHA-256, which is used in this code. Step-by-Step Process to Hash Data and Store It: Step 1: Define Data to Be Hashed: The variable data holds a sample string that will be hashed. Step 2: Convert Data to Bytes: The encode() function is used to convert the string into a byte stream (UTF-8 encoding), as the hashing algorithm works with bytes rather than strings. Step 3: Generate SHA-256 Hash:hashlib.sha256(data_bytes): This generates a hash object for the data using the SHA-256 algorithm. hexdigest(); Converts the hash object to a hexadecimal string, which is a readable format for storing and verifying the hash. Step 4: Store Data and Hash in a File: This code opens a file (hash_storage.txt) in append mode ('a') to ensure that the new data and its hash are added without overwriting any existing content. It writes the data and the corresponding hash value to the file in a human-readable format. Step 5: Print Success Message. Hash Verification Function: Verify_hash(stored_hash, new_data) is a function that compares the hash of new_data with the stored_hash.

It works by encoding the new_data into bytes, re-generating the hash for the new_data, and comparing the new hash to the stored_hash. If they match, the data has not been tampered with. Then, check data integrity. This part of the code: creating new data (new_data); using the verify_hash() function to check if the new data matches the stored hash. If the data has not changed, it prints a success message. Otherwise, it warns that the data might have been tampered with; storing data and hash using a function: This function is a reusable way to store any given data and its corresponding SHA-256 hash in the specified filename; Loading and verifying stored data: This function reads the file hash_storage.txt and checks if the given data is present. If the data is found, the function extracts the stored hash for that data and verifies it using the verify_hash() function. If the data doesn't exist in the file, it prints an error message. Finally, using load_and_verify to validate data. This calls the load_and_verify() function to check if the original data is still valid (i.e., it hasn't been altered).

5. Results

The results are shown in the following:

data: Hello, this is a sample data to hash.
hash: 5433bba1d9dbd24fab0a6ca254e12561070e5ced1fb07c172c787f6f5d7e972b
data: Sample data to hash.
hash: 2a9dc4555871092fcef8256771e6ec0a0eeaac8e0075a076c285be4230abf560

each data string has been hashed (likely using SHA-256 or a similar algorithm), producing a unique, fixed-length output (the hash). Even though the two data strings are similar, the hashes are entirely different, demonstrating the sensitivity of cryptographic hash functions to even small changes in input (this is called the "avalanche effect").

From the above results, it can be seen that the larger the amount of data stored, the more complex the hash value, and the better the encryption effect. Overall, the code implements a simple process to hash

DOI: 10.54254/2755-2721/109/2024.17894

data using SHA-256, store the data and its hash in a file, and verify the integrity of the data by comparing its hash at a later point. Storage: The store_hash() function allows you to store data and its hash in a file. Verification: The verify_hash() function checks if a new version of the data matches the previously stored hash. Validation: load_and_verify() checks if the data exists in the file and verifies its integrity. This code can be used in scenarios where data integrity needs to be preserved, such as logging, file integrity monitoring, or verifying the authenticity of messages or files.

6. Future trends

Future trends in IoT encryption technology will focus on the growing number of devices, data privacy needs, and more sophisticated cybersecurity threats. Due to the limited computing power and battery consumption of IoT devices, traditional encryption algorithms can be too complex. Lightweight encryption algorithms will gain popularity as they ensure safety while reducing processing power and energy impact. In addition, with the growing numbers of IoT devices, data transmission paths are becoming more complex. Future encryption trends will prioritize end-to-end encryption to maintain data security throughout transmission and prevent man-in-the-middle attacks. Simultaneously, as quantum computing technology advances, existing encryption methods may become vulnerable to cracking. Quantum encryption is expected to provide super-strong security for IoTs. In particular, quantum key distribution (QKD) generates unbreakable keys through the principles of quantum physics. However, securing traditional trust models (e.g., internal networks trusted, external networks untrusted) becomes challenging due to the complexity of IoT networks. A zero-trust architecture will be gradually adopted, requiring strict authentication and encryption regardless of device location on the network. Additionally, hardware-based security modules are crucial for future IoT encryption as they provide protection at the device's hardware level against physical attacks and tampering. Furthermore, blockchain technology offers a centralized trust mechanism that aligns well with the large-scale nature of IoT networks. Blockchain encryption will play a significant role in ensuring secure communications between IoT devices to safeguard data security limitations and authenticity.

7. Connection with previous research

Connection with the hybrid algorithm of MD5 and SHA-256: The combination of MD5 and SHA-256 in previous studies aimed to balance speed and security by leveraging their respective advantages. While SHA-256 significantly enhances security, it also introduces higher computational complexity. On the other hand, MD5 is fast but has well-known vulnerabilities like collision attacks. In this code, only SHA-256 is used as a purely secure algorithm without incorporating MD5, prioritizing data integrity and resistance against attacks at the expense of some speed optimization. The hybrid algorithm mentioned earlier combines the stability of MD5 with the security of SHA-256 to create a relatively faster yet secure scheme. However, this code solely focuses on security and ignores the need for speed optimization [6].

Connection to the enhanced SHA-256: The goal of studying the enhanced SHA-256 algorithm is to improve its computational efficiency for resource-based environments like embedded systems or mobile devices by optimizing processes such as loop unrolling and bit operation. The current code uses the standard SHA-256 algorithm and lacks optimization, leading to potential inefficiency in high computing resource settings or high speed requirements. Studies have shown that implementing certain optimization measures can significantly improve the computing efficiency of SHA-256 and generate devices more suitable for resource Settings. By applying these optimizations to the current code, it is able to improve its performance and generate results more appropriate for real-world use cases [6].

Connection between security and application scenarios: The core function of the code you shared is to ensure data integrity and security by generating a SHA-256 hash. SHA-256 is widely recognized for its collision resistance and protection against attacks, making it suitable for various security scenarios like digital signed certificates, encryption, blockchain, etc. Both studies emphasize the crash-resistance and temper-resistance of SHA-256, which are also the key features of this code. Specifically, the "data validation" section uses the verify_hash() function to regenerate and compare the hash value with the

DOI: 10.54254/2755-2721/109/2024.17894

stored one in order to detect any tampering with the data. This aligns perfectly with research on using SHA-256 for ensuring data integrity.

8. Conclusion

The SHA-256 algorithm is a powerful cryptographic hash function that has gained recognition for its high security and wide range of applications. Its properties, including anti-collision, anti-pre-mapping, and anti-second pre-mapping properties, make it suitable for various data security applications such as data integrity verification, digital signatures, password storage, and blockchain technology. With the proper tools and libraries, calculating SHA-256 hashes becomes simple and efficient. However, there are some drawbacks to this algorithm. First, the computational complexity of SHA-256 is high, especially for resource-constrained devices like IoT devices. Computing SHA-256 hashes can consume excessive computing resources and power in low-power environments such as embedded systems and sensor networks. Secondly, while SHA-256 is currently considered secure against classical computers' attacks; future advancements in quantum computing could pose a threat to its security. Quantum computers using the Shor algorithm can significantly accelerate asymmetric encryption attacks while Grover's algorithm can reduce the brute force break time against SHA-256 from 2^256 to 2^128 (still a very large number but compromises security). Additionally, SHA-256 is vulnerable to length extension attacks which exploit the hash function's structure allowing attackers to append data without knowing the original message when used in Message Authentication Code (MAC) protocol. Although no effective collision attack has been proven on SHA-256 yet; theoretically, speaking collision attacks on hash functions are inevitable with increasing computing power potentially leading to discovery of effective attack methods in the future (as seen with obsoletization of SHA-1 due to collision vulnerabilities). In addition, the 32-byte output of SHA-256 can be wasteful for storage-constrained environments, especially when storing a large number of hash values. This can result in unnecessary storage overhead. Although SHA-256 is secure and widely used, it has limitations in resource-constrained devices, resistance to quantum computing, storage efficiency, and certain types of attacks. Future cryptography research may develop better alternatives or enhanced variants like SHA-3 to address these challenges.

References

[1] Jing, Q., Vasilakos, A. V., Wan, J., Lu, J., & Qiu, D. (2014). Security of the Internet of Things: perspectives and challenges. Wireless networks, 20, 2481-2501.

[2] Miloslavskaya, N., & Tolstoy, A. (2019). Internet of Things: information security challenges and solutions. Cluster Computing, 22, 103-119.

[3] Borys, A., Kamruzzaman, A., Thakur, H. N., Brickley, J. C., Ali, M. L., & Thakur, K. (2022). An evaluation of IoT DDoS cryptojacking malware and Mirai Botnet. In 2022 IEEE World AI IoT Congress (AIIoT) pp. 725-729.

[4] Gowthaman, A., & Sumathi, M. (2015). Performance study of enhanced SHA-256 algorithm. International Journal of Applied Engineering Research, 10(4), 10921-10932.

[5] Kamra, S., Sharma, M., & Leekha, A. (2019). Secure hashing algorithms and their comparison. In 2019 6th International Conference on Computing for Sustainable Global Development (INDIACom) pp. 788-792.

[6] Roshdy, R., Fouad, M., & Aboul-Dahab, M. (2013). Design and Implementation a new Security Hash Algorithm based on MD5 and SHA-256. International Journal of Engineering Sciences & Emerging Technologies, 6(1), 29-36.