Implementing and attacking cryptographic hash functions

Cliford Joshy (B190539CS)[1], Joel Mathew Cherian (B190664CS)[1], and Pavithra Rajan (B190632CS)[1]

[1]Department of Computer Science and Engineering, National Institute of Technology, Calicut

Team 7

Abstract: Cryptographic hash functions are widely used for digital signatures, password tables, message-authentication codes etc. The Secure Hash Algorithm 256 (SHA-256) and Message Digest (MD5) are examples of such functions. Despite their widespread use, few cryptographers delve into the low-level implementation details of these functions.

This project aims to develop a barebone implementation of the SHA256 and MD5, using only basic programming constructs and no in-built libraries. The implementation will be tested for correctness and performance against known test vectors.

Furthermore, this project will evaluate the security of SHA256 and MD5 by simulating a length extension attack on both hashing algorithms. With this, we aim to better understand the inner workings of both algorithms and their limitations.

1 Introduction

Cryptography is a fundamental aspect of modern computer science. It plays a crucial role in protecting sensitive data in the digital world. Hash functions are a significant aspect of cryptography that has gained widespread adoption. This is primarily due to their ability to provide integrity. Additionally, it promotes authenticity in data communication. A hash function is a mathematical algorithm that takes input data of arbitrary length and produces a fixed-length output. The output produced by the hash function is referred to as a hash digest or hash value.

Primarily, a hash function generates a unique output for a given input, such that any changes to the input will result in a different hash value. This unique property of hash functions has made them useful in many areas, such as digital signatures, password storage, and data verification.

MD5 is a hash function that produces a 128-bit output. It was developed by Ronald Rivest. It is the successor to MD4. MD5 is susceptible to collision attacks with modern computers in polynomial time. MD5 is being replaced by other more secure hash functions.

SHA256 is a cryptographic hash function that produces a 256-bit output. It was developed by the National Security Agency (NSA) and is part of the SHA-2 family of hash functions. SHA256 is a secure hash function. This means it is computationally infeasible to generate the same hash value from two different inputs or find two inputs that produce the same hash value. SHA256 hash function is used to generate a checksum. A checksum is a small piece of data that is generated from a larger piece of data to ensure data integrity during transmission or storage. For instance, a user can generate the SHA256 checksum of a file before and after transmission. The two hash values should match if the file was not tampered with during transmission.

The design of MD5 and SHA256, their implementational details and their uses are discussed in upcoming sections.

2 Literature Review

All cryptographic hash functions generate a fixed-size output for a variable-sized input. Such functions can be built by repeated application of a hash function expecting fixed-size inputs. The resulting function is referred to as an iterated hash function and the fixed-size input function is called a compression function. Merkle-Damgard construction is used to construct iterated hash functions. Both the MD and SHA family of hash functions are constructed using this scheme.

2.1 Merkle-Damgard Scheme

This scheme is used to build a collision-resistant iterated hash function from a collision-resistant compression function. The compression function is a hash function that takes n-bits and m-bits fixed-size inputs. For hashing, the original message is appended with the message length and padding to reach a size divisible by n. The resulting message can thus be split into units of n-bit length. Before hashing, an m-bit digest is chosen as the initial input to the compression function, along with the first n-bit block. This digest is referred to as an initial vector or initial value. The compression function outputs an m-bit digest which is passed as input to the compression function again along with the next block of n-bits. This process repeats till all the blocks of the message are processed to give a final m-bit digest. [1]

This scheme is currently used to build many iterated hash functions. The only requirement to work with this scheme is a collision-resistant compression function. Such functions can either be built from scratch or using block ciphers. The SHA-2 and MD family of hash functions have compression functions built from scratch.

2.2 MD5

MD5 is a message digest algorithm succeeding MD4. It takes an input of arbitrary size and outputs a 128-bit fingerprint. At the time of it's release, it was computationally infeasible to find two messages with the same MD5 digest.

MD5 is an extension to MD4. It trades off speed for greater security. MD5 accepts a message of b-bits and performs the following a five steps [3]

1. Append Padding Bits: The message is padded to make it congruent to 448 modulo 512 (64 bits short of being divisible by 512). Every message passed as input to MD5 is padded. The padding is such that it starts with a single 1 and is followed by 0s.

2. Append Length: A 64-bit representation of b is appended to the message. If b is greater than 2^{64} then the lower 64 bits are used. After appending the length, the resulting message length is a multiple of 512.

3. Initialize MD buffer: Four 32-bit registers labelled A, B, C, D are initialized with specific values, and they are used for calculating MD5. These registers are the initial value in the case of MD5. During the process of MD5 the values in these registers are transformed to give the final digest.

4. Process each 512-bit block: Each 512-bit block is passed through 4 rounds. Each round includes a series of operations based on modular addition, non-linear function and left rotation. We define four auxiliary functions for this purpose, one for each round. The state of the four registers are updated after processing each 512-bit block. The updated state is used to process the next block. This iteration then continues.

5. Output: The message digest generated at the end is A, B, C, D.

2.3 SHA256

It uses six logical functions to operate on 32-bit words to generate a new 32-bit word. SHA-256 uses 64 constants which are 32-bit words. These are the first 32 bits of the fractional part of the cube root of the first 64 prime numbers. SHA256 hashing is permissible on message size with length $< 2^{64}$. The following 4 steps take place in SHA256 hashing [2]:

1. Padding the message: Pad the message similarly to MD5. The length of the padded message must be congruent to 448 modulo 512. The remaining 64 bits are for the binary representation of the length of the original message. Here also, there is at least one padding. It starts with 1 followed by k bits of 0s such that it adheres to the aforementioned congruency rule.

2. Parsing Message: Converting the padded message to 512 bits blocks.

3. Initialization of the hash values: The initial values are eight 32-bit words. These are the first 32 bits of the fractional parts of the square roots of the first eight prime numbers.

4. Processing: We prepare a message schedule using logical functions and 32-bit blocks. We then use these to create a series of operations involving auxiliary functions and logical operations along with constants to update the initial eight 32-bit numbers into the hash. The final hash size is 256bit.

2.4 Comparing SHA256 and MD5

MD5 generates a message digest of 128-bit length, while SHA256 generates a hash of 256-bit length. Due to its larger size, SHA256 is more collision resistant. However, SHA256 is slower than MD5 in computation. SHA256 finds wide application today in blockchain, checksum, etc. Use of MD5 in applications is discouraged since it is not secure anymore.

2.5 Attacks on hashing algorithms

There are two categories of attacks under this domain

- **Generic attacks:** Attacks for all hash functions
- **Specific attacks:** Attacks crafted for a specific hash function

Given a hash function and a particular hash, obtaining an input that produces the said hash requires an exhaustive search. Arbitrarily chosen inputs are passed to the hash function in an attempt to obtain a particular hash. Getting a hit would take $2^{(n-1)}$ such arbitrary evaluations. On the other hand, finding a collision is much easier. This result is an implication of the birthday paradox. To find a collision $1.2{*}2^{n/2}$ evaluations is sufficient. A hash function is ideal if the best attacks on it are generic. Additionally, a hash function is known to be broken if its security for one of its three properties is below ideal security. The three properties of concern are [4]:

- **One-wayness:** The function must be one-way and not reversible. Ideally, it must take 2^{n-1} attempts to break this.
- **Second pre-image resistance:** Given a hash and a hash function finding a pre-image must be computationally hard. Ideally, it must take 2^{n-1} attempts to break this.
- **Collision resistance:** Finding two inputs mapped to the same hash should be computationally hard. Ideally, it must take $1.2{*}2^{n/2}$ attempts to break this.

Iterated hash functions constructed using the Merkle Damgard Scheme also suffer from several attacks due to the nature of their construction. While the literature on this topic is broad, this paper focuses on the case of Length Extension Attack (LEA) alone.

2.6 Length Extension Attack (LEA)

LEA is a type of attack on certain hash functions that allows an attacker to append additional data to a given hash value and compute the hash of the new message without knowing the original message. This can be a problem if the hash value is being used as a message authentication code or a digital signature, as the attacker can create a valid-looking message that was not actually signed or authenticated by the original sender.

2.7 Applications of SHA256

SHA256 has various applications in the digital world. Some of the applications of SHA256 are discussed below [4]:

- **Password Storage:** SHA256 is used in password storage to ensure that passwords are not stored in plaintext. When a user creates a password, the password is hashed using SHA256, and the hash value is stored in a database. When the user enters their password during login, the entered password is hashed using SHA256, and the resulting hash value is compared with the hash value stored in the database. If the hash values match, the user is granted access.

- **Digital Signatures:** Digital signatures are used to authenticate the origin of data and ensure that data has not been tampered with during transmission. SHA256 is used in digital signatures to generate a hash value of the data to be signed. The hash value is then encrypted using the sender's private key to generate a digital signature. The recipient can then use the sender's public key to decrypt the digital signature and verify the authenticity of the data.
- **Blockchain:** Blockchain is a distributed ledger technology used in cryptocurrencies and other applications that require a tamper-proof record of transactions. SHA256 is used in blockchain to generate a unique hash value for each block in the blockchain. The hash value of each block is used to link it to the previous block, creating a chain of blocks that cannot be tampered with.

3 System/Network environment

The system requirements are as follows:

- **Operating System:** SHA256 implementation can be done on any operating system, including Windows, Linux, and MacOS.
- **Python 3:** The implementation is done using Python. You will need Python 3 installed on the system. Python3 is the latest version of Python and comes with many improvements over the previous version.
- Required Python Modules: You will need to have the following Python modules installed:
  - math: This is already pre-installed with the latest Python version
  - argparse
  - logging
- **Memory:** Sufficient memory is required to store the input message and intermediate hash values.
- **Disk Space:** There are no significant disk space requirements.
- There are no network requirements for this implementation. However, you will need internet connectivity for installing Python and the required packages via package managers like pip or conda.

4 Design

In the above design, the file for which we need to attain the SHA256 hash is read. The contents of the file is then encoded for further processing. The encoded message is then padded by adding 1 bit to the end followed by bits of 0's such that the total length attained so far is congruent to 448 module 512. Following this, 64-bit binary representation of the original encoded message is concatenated. Once the padding is completed successfully, it is parsed to create blocks of 512-bit. The hash values are then initialised, followed by message schedule generation. Through a series of auxiliary functions, the working variables are modified to compute the hash values. The final eight 32-bit hash values, thereby obtained, are then concatenated to attain the 256-bit hash.

5 Progress of implementation

Through the course of this project, the following work has been done:

- Implemented the SHA256 algorithm without using any built-in libraries. The implementation was done using the Python programming language, and we have ensured that the implementation is correct by adhering to the NIST standards for SHS. `https://github.com/Pavithra-Rajan/BareSHA-256`
- Created a bash script for testing the implementation against the sha256sum command line utility. This script was used to check

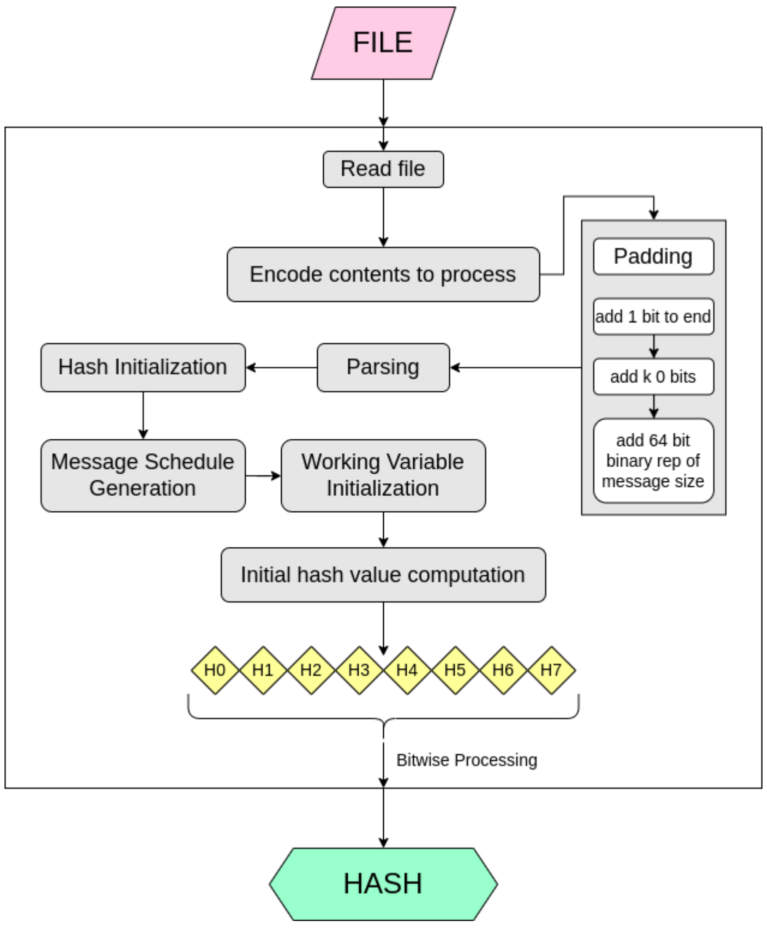
FILE
Read file
Encode contents to process
Padding
add 1 bit to end
add k 0 bits
add 64 bit binary rep of message size
Hash Initialization
Parsing
Message Schedule Generation
Working Variable Initialization
Initial hash value computation
H0
H1
H2
H3
H4
H5
H6
H7
Bitwise Processing
HASH

Figure 1: SHA256 Design

if the checksum created by our implementation is the same as the one generated by sha256sum. The script also computed the execution time taken by our implementation.

- Conducted an extensive literature survey on hash functions, SHA, and MD family, along with their vulnerabilities. This helped us to understand the intricacies of hash functions and to design our implementation accordingly.
- Created the design and extensive documentation of our learnings throughout the implementation process. This documentation included detailed explanations of the SHA256 algorithm, the NIST standard for SHS, the implementation process, the testing process, and the results obtained.

6 Conclusion

Implementing the SHA256 checksum from scratch without using any built-in libraries gave us a deep understanding of the intricacies of cryptographic hash functions and the steps involved in computing the SHA256 checksum. Despite its widespread use, there is a lack of understanding about how SHA256 and other cryptographic hash functions work. This is partly because many people rely on pre-built libraries and tools that handle the implementation details but also because the underlying mathematics can be complex and challenging to understand.

Through this project, we learned about the various components of the SHA256 algorithm, including message padding, parsing and message schedule

```
$ python3 sha256.py -h

    usage: sha256.py [-h] -f F

    options:
      -h, --help  show this help message and exit
      -f F        Name of the file to find the checksum


$ python3 sha256.py -f tests/test1.pdf

    7e2903a8c60bf957824c330707617e8cc32283b5287bb9362acb2f45550810c1


$ ./test.sh

    Passed: tests/test1.pdf
    Passed: tests/test2.txt
    Passed: tests/Nessus.deb
```

Listing 1: Running Barebone SHA256

computation. We were able to see first-hand how the various components of the algorithm work together to produce a secure and reliable hash value. Additionally, we gained a better appreciation for the underlying mathematics and logic behind cryptographic hash functions. This project was an excellent opportunity for us to delve deeper into the world of cryptography and gain practical experience in implementing a widely-used hash function.

References

[1] Forouzan, B. A., & Mukhopadhyay, D. (2015). Cryptography and network security (Vol. 12). New York, NY, USA:: Mc Graw Hill Education (India) Private Limited.

[2] NIST. (2012, March). Secure Hash Standard. NIST Secure Hash Standards. Retrieved March 25, 2023, from https://csrc.nist.gov/csrc/media/publications/fips/180/4/final/documents/fips180-4-draft-aug2014.pdf

[3] IETF. (1992, April). RFC 1321. IETF RFC. https://www.ietf.org/rfc/rfc1321.txt

[4] Mironov, I. (2005). Hash functions: Theory, attacks, and applications. Microsoft Research, Silicon Valley Campus, 1-22.