|  |
| --- |
| Group 11  Mona Shah ms4502@nyu.edu  Stephen Seeler stephen.seeler@nyu.edu  Arul Yagappan aty2016@nyu.edu  Will DeSantis wld233@nyu.edu |

Applied Crypto

CS-GY 6903 CF01/CF02

**Project4: Cryptographic Hash**

**Part I: Literature Review & Part II: Discussion Questions**

Please refer to the document “**Project4 \_Problems 1 and 2**”

**Part III: Code Project**

1. Merkle Tree Implementation

Please see **3.1.py** and **test\_file[1-6].txt**

Relevant comments are included in the code.

Execution screenshot below:

A screenshot of a computer

Description automatically generated

1. Root Hash Observation

**Root Hash Change:** Modifying the content of one text file will change its hash value. Since the Merkle tree is constructed based on the hash values of the leaf nodes, changing the content of one leaf node will affect its hash value and consequently affect the hash values of its parent nodes up to the root.

**Different Root Hash:** After modifying the content of one leaf node, the root hash of the Merkle tree will be different from the original root hash. This is because the root hash is computed based on the hash values of all leaf nodes, and any change in a leaf node's content will propagate changes up to the root.

**Verification Failure:** If you compare the new root hash with the original root hash, they will not match. This indicates that the integrity of the Merkle tree has been compromised due to the change in the leaf node's content. This verification failure demonstrates the security feature of Merkle trees in detecting tampering or modifications in the data.

1. Hash Collision

Please see **3.3.py** and **file\_[1-6].txt**

Relevant comments are included in the code.

Execution screenshot shown below:

A screenshot of a computer program

Description automatically generated

**Strategies:** Finding collisions with hashes of varying lengths requires different strategies depending on the length of the hash. Here are a few strategies for finding collisions with hashes ranging from 4 bits to 160 bits:

1. Brute Force: Suitable for very short hashes (e.g., 4 bits), systematically generate different inputs until finding two that produce the same hash.

2. Birthday Attack: Exploit the birthday paradox to find collisions efficiently, becoming more effective as the hash length increases.

3. Meet-in-the-Middle Attack: Precompute hashes for a large set of inputs and compare against hashes generated for a different set of inputs to find collisions more efficiently.

4. Parallelization: Distribute the workload across multiple computing nodes or GPUs to speed up collision finding, especially for longer hash lengths.

5. Differential Cryptanalysis: Analyze differences in hash outputs for similar inputs to identify patterns or vulnerabilities in the hash function.

6. Cryptanalysis Techniques: Exploit weaknesses in the hash function's compression function or algebraic properties to find collisions.

7. Hybrid Approaches: Combine multiple strategies, such as brute force with parallelization or differential cryptanalysis with a birthday attack, for enhanced efficiency.

8. Optimization Techniques: Optimize search algorithms, data structures, and memory management to improve collision finding performance.

1. Hash Puzzle

Please see **3.4.py**

Relevant comments are included in the code.

Execution Screenshot below:

A screen shot of a computer

Description automatically generated

**Solving a puzzle requiring a 20-bit zero prefix for SHA256** involves a significant workload due to the nature of cryptographic hash functions and the size of the search space. Overview of the workload involved:

1. Search Space: SHA256 produces a 256-bit hash output. Finding a hash with a 20-bit zero prefix means searching through 220220 possible nonce values until a suitable hash is found.
2. Brute Force: The search process typically involves incrementing a nonce value, hashing it using SHA256, and checking if the hash has the required zero prefix. Since each nonce value needs to be hashed and checked individually, it's a brute-force approach.
3. Time Complexity: The time complexity of finding a hash with a 20-bit zero prefix depends on the computational power available. It could range from feasible to impractical, depending on the hardware resources and time constraints.
4. Computational Resources: Solving such puzzles may require significant computational resources, including CPU time and memory. The workload increases exponentially as the number of required zero bits grows larger.
5. Parallelization: To reduce the time required to solve the puzzle, parallelization techniques can be employed, utilizing multiple computing nodes or GPU-based processing to search the nonce space concurrently.
6. Energy Consumption: The workload also translates into energy consumption, especially for large-scale mining operations. Solving puzzles with stringent requirements may consume considerable electrical power.