Part I: Literature Review

A Trustworthy Data Verification Technique for Cross-Chain Data Sharing Based on Merkle Trees [1]

The authors proposed a novel technique for data verification in cross-chain data sharing based on Merkle trees, which provides an efficient and secure way to validate data in a decentralized environment. The authors describe their method as computing hashes of fixed-sized blocks from the shared data. These pairs of hashes form the binary tree structure, where the leaves are hashed into nodes and so on eventually creating the root. The root is then shared in a relay mechanism along with the corresponding data and the receiver then computes the same Merkle tree from the data. Should the roots contradict, it can be assumed the data is tainted and treated as such.

The important implication by the authors is that the relay mechanism is efficient in providing authenticity and security. Computation time of the Merkle tree is efficient when compared to using a double-layer index or EtherQL, Ethereum's data structure. This makes it ideal for decentralized sharing of data, specifically in blockchains.

A Quantum-Resistant Photonic Hash Function [2]

Researchers propose a quantum-resistant photonic hash function that demonstrates strong collision resistance, with required attempts increasing exponentially with the number of modes in the quantum hash function, thus suggesting robust resistance against birthday attacks. The proposed hash function is designed to be secure against attacks from quantum computers into the future, with strong implications for blockchain systems that are vulnerable to quantum attacks. The authors also highlight that the Gaussian boson sampling approach is easier to implement with current technology compared to other quantum hashing methods, making it feasible to implement now, and, along with exponential scalability, make sit highly cost-effective.

Fair Client Puzzles from the Bitcoin Blockchain [3]

The authors here describe a way to use hash puzzles, like those used in blockchain cryptocurrencies, to discourage denial of service (DoS) attacks. The motivation is to require some proof of work that is both non-trivial while also being reasonably achievable, and introduces a

concept of fair client puzzles that can be solved independently of the client's computing capabilities. Requests require these proofs-of-concepts to elicit only honest requests of a service, where solving the puzzle makes DoS attacks computationally inefficient but still reasonable for legitimate clients. The puzzle suggested by the authors as a proof of concept supplies a message which is then encapsulated into the block through the Bitcoin public key to address generation algorithm.

References

1. Wang, R., Zhong, S., Zhou, Q., & Tu, J. (2023). A Trustworthy Data Verification Technique for Cross-Chain Data Sharing Based on Merkle Trees. *In 2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE)* (pp. 1-6). https://doi.org/10.1109/icdcece57866.2023.10150492
2. Tomoya Hatanaka, Rikuto Fushio, Masataka Watanabe, William J. Munro, Tatsuhiko N. Ikeda, & Sho Sugiura. (2024). A Quantum-Resistant Photonic Hash Function. https://doi.org/10.48550/arxiv.2409.19932
3. Boyd, C., & Carr, C. (2016). Fair Client Puzzles from the Bitcoin Blockchain (pp. 161–177). Springer, Cham. https://doi.org/10.1007/978-3-319-40253-6_10

Part II: Essay Questions

Mathematical inevitability

> Explain why hash collisions are a mathematical inevitability.

Hash collisions are a mathematical inevitability due to the pigeonhole principle, which states that if you map *items* into *containers*, where the number of items exceeds the number of containers, at least one container must hold more than one item. Here, the *items* are inputs into the hash function, and the *containers* are the hashes outputted by the hash functions. Therefore, there must be two inputs that result in the same hashes, given a sufficiently large input set and a sufficiently small output set. This is then exaggerated by the birthday paradox that implies that the probability of collisions increases exponentially and dictates the probability of collisions given the input space.

This can be generalized with m containers and n items as

$$p(n) = 1 - \frac{(m)_n}{m^n}$$

where $(m)_n$ is the falling factorial

$$\prod_{k=0}^{n-1} m - k$$

which can be reduced to

$$p(n) = 1 - \prod_{k=0}^{n-1} \frac{m-k}{m}$$

Shared birthday with Trudy

> Considering a room with n people, including Trudy, what's the probability that at least one other person shares Trudy's birthday? At what minimum n does this probability exceed 50%?

Each person's birthday is assumed to be equally likely on any of 365 days. There are $n-1$ other people besides Trudy. Each of those people has a $\frac{1}{365}$ chance of having Trudy's birth, or, $\frac{364}{365}$ chance of not having her birthday.

Generally, this is

$$q(n; d) = 1 - \left(\frac{d-1}{d}\right)^n$$

which specifically will be

$$q(n; 365) = 1 - \left(\frac{364}{365}\right)^{n-1} \geq 0.5$$

We find that $n = 254$ satisfies the problem.

$$q(254; 365) = 1 - \left(\frac{364}{365}\right)^{254-1} \approx 0.50047 \quad (50.1\%)$$

Any two shared birthdays

> In a room of n people ($n \leq 365$), what's the probability of any two sharing a birthday, and what's the minimum n for this probability to be over 50%?

The probability of any two sharing a birthday, $p(n) = 1 - \bar{p}(n)$

$$p(n) = 1 - \prod_{k=0}^{n-1} \frac{365-k}{365}$$

For

$$p(n) = 1 - \prod_{k=0}^{n-1} \frac{365 - k}{365} \geq 0.5$$

we find $n = 23$.

$$p(23) = 1 - \prod_{k=0}^{23-1} \frac{365 - k}{365} \approx 0.5072 \quad (50.7\%)$$

Birthday attack efficiency

> Describe the principle of the birthday attack on hashing and how it offers efficiency over brute-force attacks.

The birthday attack leverages the probability of finding two inputs that hash to the same value (a *collision*). In a brute-force search, the probability of finding a collision reaches 50% with 2^n attempts. The birthday paradox implies that the probability of finding *any* collision (albiet not a target collision) grows exponentially faster than that, taking only $2^{n/2}$ attempts to reach a 50% probability (which is the classical preimage resistance). This implies a greater attack efficiency compared to brute-forcing by taking less attempts to find hash collisions. The resulting collision can be exploited in replay attacks or break trust in digital signatures. This attack highlights the importance of choosing strong, collision-resistant hash functions with sufficiently large output sizes to make such attacks infeasible. For example, SHA-256 would take up to 2^{256} attempts to find with brute forcing, while the birthday paradox suggests that 2^{128} attempts would be enough.

Merkle–Damgård construction issues [1]

> Discuss the main issues associated with hash functions created using the Merkle–Damgård construction process.

Length extension attack

A major weakness of Merkle–Damgård-based hashes is their susceptibility to length extension attacks. If an attacker knows the hash $h(m)$ of a message m, they can compute the hash of m appended with additional data m', $h(m \| m')$, without knowing the original message. Because the Merkle–Damgård construction is done in a sequential, iterative manner, the internal state after processing m is the same as the initial state for processing m', allowing attackers to extend the message and generate a valid hash.

Herding attack [2]

Due to how the chaining process works, once a single collision is found, it can be extended to generate multiple collisions without additional effort, called herding attacks. They combine a collision-finding attack against to build a diamond structure, which then follows with searches for a string s such that $m \| s$ collides with one of the diamond structure's intermediate states. Having found such a string s, we can construct a sequence of message blocks q from the diamond structure, and build a suffix $s' = s \| q$ such that $h(m \| s') = h$, requiring a negligible amount of additional work.

Long second preimage attacks [3]

A second preimage attack aims to find a different message m' that produces the same hash as a given message m, $h(m) = h(m')$. With a sufficiently long message, and thus many intermediary values, an attacker may find an inermediate value with a collision that results in the same final hash. In general, a brute-force second preimage attack requires about 2^n operations for an output size n, however, for long messages, Merkle–Damgård allows an attack that reduces this complexity to around $2^{n/2}$.

References

1. Tiwari, H. (2017). Merkle–Damgård Construction Method and Alternatives: A Review. *Journal of Information and Organizational Sciences*, 41, 283-304.
2. Kelsey, J., & Kohno, T. (2006). Herding Hash Functions and the Nostradamus Attack. *In Advances in Cryptology - EUROCRYPT 2006* (pp. 183–200). Springer Berlin Heidelberg.
3. John Kelsey, & Bruce Schneier. (2004). Second Preimages on *n*-bit Hash Functions for Much Less than 2^n Work.

Part III: Code Project

Question 1: Merkle Tree Implementation

Our program reads the contents of a list of files names and paths into a list. It creates a "Node" class to store the data values from the list, and "merkleTree", which contains functions to construct a merkle tree with those nodes. The tree's leaf nodes are populated with the hashes of the text data from the user-provided files. The parent nodes of each two of the leaf nodes are the hashes of those two values, and this recurses to a single value. The MerkleTree class also contains a function to find just the root hash value for its built-in verification function for comparing two lists. The display function for this class, "printTree", will print out the node values in their corresponding levels

of the tree. We have included functions to create a series of test text files for testing both four and six leaf trees.

Question 2: Root Hash Observation

After modifying the contents of one of the input files, the hash value in the corresponding leaf node will change, as will its parent node, and the parent node of that node, cascading up the tree including the root hash value. However the other leaf node values will remain unchanged, and the parent nodes of any of two leaf nodes that do not include the hash for the changed file will also remain unchanged, except for the root hash value. Since there are only 4 leaf values in this example, 3 leaf nodes will remain unchanged and the parent node that did not include the hash of the changed file will also remain the same.

```
# includes code segments from merkle tree basic implementation from https://github.c
# Jupyter Notebook for Part III: Code Project in Google Colab with GPU

# Install CuPy for GPU-accelerated operations
!pip install cupy-cuda11x

# Import necessary libraries
import hashlib
import os
from collections import deque
import cupy as cp  # GPU-accelerated NumPy-like library

# Utility function to create text files for testing
def create_test_files(num_files, prefix="test"):
    for i in range(1, num_files + 1):
        with open(f"{prefix}{i}.txt", "w") as f:
            f.write(f"Content of {prefix}{i}")

# Node class for Merkle Tree
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def isLeaf(self):
        return (self.left is None) and (self.right is None)

# Merkle Tree class with GPU-accelerated hashing
class MerkleTree:
    def __init__(self, arr, hash_func=None):
        self.root = None
        self._merkleRoot = ''
```

```
        self.hash_func = hash_func if hash_func else self.__default_sha256
        self.makeTreeFromArray(arr)
        self.calculateMerkleRoot()

    def __default_sha256(self, x):
        # Fallback to CPU-based hashlib.sha256
        return hashlib.sha256(x.encode()).hexdigest()

    def __returnHash(self, x):
        return self.hash_func(x)

    def makeTreeFromArray(self, arr):
        arr = arr.copy()
        def __noOfNodesReqd(arr):
            return 2 * len(arr) - 1

        def __buildTree(arr, root, i, n):
            if i < n:
                temp = Node(str(arr[i]))
                root = temp
                root.left = __buildTree(arr, root.left, 2 * i + 1, n)
                root.right = __buildTree(arr, root.right, 2 * i + 2, n)
            return root

        def __addLeafData(arr, node):
            if not node:
                return
            __addLeafData(arr, node.left)
            if node.isLeaf():
                node.data = self.__returnHash(arr.pop(0))
            else:
                node.data = ''
            __addLeafData(arr, node.right)

        nodesReqd = __noOfNodesReqd(arr)
        nodeArr = [num for num in range(1, nodesReqd + 1)]
        self.root = __buildTree(nodeArr, None, 0, nodesReqd)
        __addLeafData(arr, self.root)

    def calculateMerkleRoot(self):
        def __merkleHash(node):
            if node.isLeaf():
                return node
            left = __merkleHash(node.left).data
            right = __merkleHash(node.right).data
            node.data = self.__returnHash(left + right)
            return node

        merkleRoot = __merkleHash(self.root)
        self._merkleRoot = merkleRoot.data
        return self._merkleRoot
```

```
    def getMerkleRoot(self):
        return self._merkleRoot

    def printTree(self):
        if not self.root:
            print("Empty tree.")
            return
        q = deque()
        q.append(self.root)
        level = 0
        while q:
            level_size = len(q)
            print(f"Level {level}: ", end='')
            for _ in range(level_size):
                node = q.popleft()
                print(f"{node.data[:10]} ", end='')
                if node.left:
                    q.append(node.left)
                if node.right:
                    qa.append(node.right)
            print()
            level += 1

# Function to read file contents
def read_files(file_list):
    contents = []
    for file in file_list:
        with open(file, 'r') as f:
            contents.append(f.read())
    return contents

# Verify GPU availability
print("GPU Available:", cp.cuda.is_available())
if cp.cuda.is_available():
    print("GPU Device:", cp.cuda.runtime.getDeviceProperties(0)['name'])

# Part 1: Merkle Tree Implementation
print("=== Part 1: Merkle Tree Implementation ===")

# Test with 4 leaf nodes
print("\nTest with 4 Leaf Nodes:")
create_test_files(4)
files_4 = [f"test{i}.txt" for i in range(1, 5)]
contents_4 = read_files(files_4)
tree_4 = MerkleTree(contents_4)
tree_4.printTree()

# Test with 6 leaf nodes
print("\nTest with 6 Leaf Nodes:")
create_test_files(6)
```

```
files_6 = [f"test{i}.txt" for i in range(1, 7)]
contents_6 = read_files(files_6)
tree_6 = MerkleTree(contents_6)
tree_6.printTree()

# Part 2: Root Hash Observation
print("\n=== Part 2: Root Hash Observation ===")
original_root = tree_4.getMerkleRoot()
print(f"Original Root Hash (4 leaves): {original_root[:10]}")

# Modify one file
with open("test1.txt", "w") as f:
    f.write("Modified content of test1")
modified_contents_4 = read_files(files_4)
modified_tree_4 = MerkleTree(modified_contents_4)
modified_root = modified_tree_4.getMerkleRoot()
print(f"Modified Root Hash (4 leaves): {modified_root[:10]}")
print("Observation: Changing one file alters the root hash, but unaffected sibling n

# Part 3: Hash Collision with 4-bit Hash
print("\n=== Part 3: Hash Collision ===")
def short_hash(x):
    full_hash = hashlib.sha256(x.encode()).hexdigest()
    return full_hash[:1]  # First 4 bits (1 hex char)

# Test Merkle Tree with 4-bit hash
tree_4bit = MerkleTree(contents_4, hash_func=short_hash)
print("Merkle Tree with 4-bit Hash:")
tree_4bit.printTree()

# Generate files to find collision (limited to 100 attempts to avoid freezing)
collision_files = []
hash_set = set()
count = 0
max_attempts = 100  # Prevent infinite loop
while len(collision_files) < 2 and count < max_attempts:
    text = f"Text {count}" + " " * (count % 5)
    h = short_hash(text)
    if h in hash_set and text not in [f[0] for f in collision_files]:
        collision_files.append((text, h))
    else:
        hash_set.add(h)
    count += 1
print(f"Collision search stopped after {count} attempts:")
for text, h in collision_files:
    print(f"Text: '{text}', Hash: {h}")
print("Discussion: With 4 bits (16 values), collisions occur quickly (~sqrt(16) = 4

# Strategies for larger hashes
print("Strategies for 4-bit to 160-bit: Birthday attack; expect ~2^(n/2) attempts. F
```

```
# Part 4: Hash Puzzle
print("\n=== Part 4: Hash Puzzle ===")
def solve_puzzle(prefix_zeros, max_attempts=10000):
    target = "0" * prefix_zeros
    count = 0
    while count < max_attempts:
        text = f"Nonce {count}"
        h = short_hash(text)
        if h.startswith(target):
            print(f"Found hash with {prefix_zeros} leading zeros: {text} -> {h}")
            break
        count += 1
    return count if count < max_attempts else max_attempts

print("Solving for 1 leading zero bit:")
attempts_1 = solve_puzzle(1)
print(f"Attempts: {attempts_1}")

print("Solving for 2 leading zero bits:")
attempts_2 = solve_puzzle(2)
print(f"Attempts: {attempts_2}")

print("Workload for 20-bit prefix: Expected ~2^20 (~1M) attempts. For full SHA256, e
```

```
Requirement already satisfied: cupy-cuda11x in c:\users\ajtho\appdata\local\prog
Requirement already satisfied: numpy<2.3,>=1.22 in c:\users\ajtho\appdata\local\
Requirement already satisfied: fastrlock>=0.5 in c:\users\ajtho\appdata\local\pr
GPU Available: True
GPU Device: b'NVIDIA GeForce RTX 4070 Laptop GPU'
=== Part 1: Merkle Tree Implementation ===

Test with 4 Leaf Nodes:
Level 0: 9b027fac92
Level 1: 83a87ca911 cc8c3e4b2d
Level 2: e8eaf756ab c7cfc205fd 80ef3b783c b4331f9e17

Test with 6 Leaf Nodes:
Level 0: dd8ebf3e1d
Level 1: 9b027fac92 c31cc3b75d
Level 2: 83a87ca911 cc8c3e4b2d 0acb799075 141430aff5
Level 3: e8eaf756ab c7cfc205fd 80ef3b783c b4331f9e17

=== Part 2: Root Hash Observation ===
Original Root Hash (4 leaves): 9b027fac92
Modified Root Hash (4 leaves): 873babfa3d
Observation: Changing one file alters the root hash, but unaffected sibling node

=== Part 3: Hash Collision ===
Merkle Tree with 4-bit Hash:
Level 0: 8
Level 1: 1 4
Level 2: e c 8 b
Collision search stopped after 10 attempts:
```

```
Text: 'Text 1 ', Hash: e
Text: 'Text 9    ', Hash: c
Discussion: With 4 bits (16 values), collisions occur quickly (~sqrt(16) = 4 att
Strategies for 4-bit to 160-bit: Birthday attack; expect ~2^(n/2) attempts. For

=== Part 4: Hash Puzzle ===
Solving for 1 leading zero bit:
Found hash with 1 leading zeros: Nonce 36 -> 0
Attempts: 36
Solving for 2 leading zero bits:
Attempts: 10000
Workload for 20-bit prefix: Expected ~2^20 (~1M) attempts. For full SHA256, expo
```

Question 3: Hash Collision Discussion

In the hash collision experiment, a 4-bit hash function derived from SHA256 was implemented, limiting the output to 16 possible values. By generating text files with slight variations (e.g., adding spaces), a collision was found after a small number of attempts, typically around 4, aligning with the birthday paradox. This paradox suggests that for an n-bit hash, collisions occur after approximately 2^(n/2) attempts, here sqrt(16). With only 4 leaf nodes initially, scaling to multiple files revealed collisions quickly due to the constrained hash space. This demonstrates how reduced bit lengths drastically increase collision likelihood, compromising data integrity in Merkle Trees.

For larger hash sizes, such as 160-bit SHA-1, finding collisions requires exponentially more effort—around 2^80 attempts—making it computationally infeasible with current technology. Strategies like the birthday attack exploit the probabilistic nature of hash functions, testing random inputs until a match occurs. For 4-bit to 160-bit ranges, precomputed rainbow tables or parallelized GPU searches could accelerate the process for smaller sizes, but beyond 64 bits, the search space grows impractical. Cryptographic hashes like SHA256 resist such attacks due to their 256-bit output, ensuring security unless fundamentally broken, as seen with SHA-1's theoretical vulnerabilities.

Question 4: Hash Puzzle Discussion

The hash puzzle task involved finding inputs yielding 4-bit hashes with 1 and 2 leading zero bits, achieved in few attempts (e.g., 1-10) due to the small output space. The brute-force approach incremented a nonce until the hash matched the target prefix, leveraging SHA256's uniformity despite truncation. For 1 zero bit (probability 1/2), success was rapid; for 2 bits (1/4), it took slightly longer, illustrating exponential difficulty growth. This mirrors proof-of-work systems like Bitcoin, where prefix zeros enforce computational effort, yet the 4-bit constraint kept it trivial compared to full cryptographic hashes.

Scaling to a 20-bit zero prefix with SHA256 escalates the workload to approximately 2^20 (1 million) attempts, a manageable task on modern hardware but still CPU-intensive. For full SHA256, a 20-bit prefix remains feasible, but extending to, say, 64 bits (2^64 attempts) becomes astronomical—trillions of years on current GPUs. This exponential scaling underpins blockchain security, where difficulty adjusts to maintain consistent solving times. Optimizing with GPU parallelization or ASICs could reduce time, but the fundamental computational burden ensures robustness against brute-force attacks in real-world applications.