Project: Cryptographic Hash

Part I: Literature Review

A literature review refers to a comprehensive survey of existing research, studies, and scholarly articles related to specific topics. In this project, students are asked to familiarize themselves with the topics on Merkle trees, hash collisions, and hash puzzles. Students are expected to identify and analyze sources that discuss these concepts and their recent developments.

Requirements:

• Compile a reference list of at least three sources that provide insight into the concepts of Merkle trees, hash collisions, hash puzzles, and their recent development.

• For each reference, summarize the main ideas in a brief paragraph.

• Sources can include book chapters, peer-reviewed journal articles, conference papers, or credible internet sources.

• The following references are provided as starting points but must not be used in your list:

• B. Weber and X. Zhang. Parallel hash collision search by rho method with distinguished points. Proc. of the 14th IEEE LISAT 2018, Farmingdale, NY, May 4, 2018, pp. 1-7.

• Mark Stamp's book (as listed in the course syllabus), specifically sections 5.2 (Birthday attack, Nostradamus attack) and 5.3 (MD4). • J. Kelsey and T. Kohno, Herding hash functions and the Nostradamus attack, eprint.iacr.org/2005/28l.pdf.

Source 1: Merkle Trees

Title: Merkle trees in blockchain: A Study of collision probability and security implications
Authors: Oleksandr Kuznetsov, Alex Rusnak, Anton Yezhov, Kateryna Kuznetsova, Dzianis Kanonik, Oleksandr Domin
Publication: Internet of Things, Volume 26, 2024, 101193, ISSN 2542-6605
Reference: O. Kuznetsov, A. Rusnak, A. Yezhov, K. Kuznetsova, D. Kanonik, and O. Domin, "Merkle trees in blockchain: A study of collision probability and security implications," Internet of Things, Elsevier, Jul. 2024.

Summary:
This study examines the security of Merkle Trees, a core component of blockchain systems like Ethereum, focusing on their vulnerability to hash collisions and preimage attacks. Through theoretical and empirical analysis, the researchers assess how hash length and path length influence collision probabilities. Findings reveal that longer hash lengths enhance security, while longer paths increase collision risks. These insights help improve blockchain resilience, particularly in applications like XMSS digital signatures, which rely on Merkle Trees for quantum-resistant security.

Source 2: Hash Collisions

Title: Finding Collisions in the Full SHA-1
Authors: Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu
Publication: CRYPTO 2005: Advances in Cryptology
Reference: Wang, X., Yin, Y. L., & Yu, H. (2005). Finding Collisions in the Full SHA-1. CRYPTO 2005: Advances in Cryptology.

Summary:
This groundbreaking work demonstrates the first practical cryptanalysis of full SHA-1, achieving collisions below the theoretical security threshold through three key innovations: (1) a novel disturbance vector optimization technique that relaxes traditional constraints to uncover low-probability differential paths, (2) strategic exploitation of arithmetic carries and Boolean function properties to control difference propagation, and (3) an efficient two-block attack framework that converts near-collisions into full collisions. By reducing the attack complexity to 2^{69} operations - significantly beneath SHA-1's 2^{80} design strength - and experimentally validating the approach with 58-step collisions (2^{33} complexity), the research exposes fundamental vulnerabilities in SHA-1's structure, directly contributing to its deprecation in cryptographic applications and reshaping modern hash function security standards.

Source 3: Hash Puzzles

Title: Proofs of Work and Bread Pudding Protocols
Authors: Markus Jakobsson and Ari Juels
Publication: Springer, Boston, MA (Print ISBN: 78-1-4757-6487-1)

Summary:

This paper formalizes proofs of work (PoW) as cryptographic protocols where a prover verifiably demonstrates computational effort—rather than secret knowledge—and introduces bread pudding protocols, which repurpose PoW computations for useful tasks (like MicroMint coin minting). By breaking intensive operations into parallelizable PoWs, the authors show how "wasted" work can be transformed into productive outputs while maintaining security, laying early groundwork for blockchain consensus and decentralized trust mechanisms.

Part II: Essay Questions

1) Explain why hash collisions are a mathematical inevitability?

Think of a hash function as a clever chef who takes ingredients—inputs of all shapes and sizes—and whips them into a dish of fixed portions, like the 256-bit servings you get from SHA-256. But here's the catch: sometimes, totally different recipes end up tasting the same. That's what we call a hash collision, and it's not a fluke—it's baked into the math. The Pigeonhole Principle is the kitchen rule here: if you've got more ingredients than plates to serve them on, some dishes are bound to double up. Sure, 2^256 possible hash outputs sounds like a feast, but it's still a limited menu compared to the endless pantry of possible inputs. So, no matter how big the table, you can't escape it—collisions are just part of the deal when you're squeezing an infinite world into a finite box.

Mathematically: Hash collisions are inevitable due to the Pigeonhole Principle, as an infinite input space ($|I| \to \infty$) maps to a finite output space ($|O| = 2^n$), ensuring at least one pair of distinct inputs ($x \neq y$) where $H(x) = H(y)$ for any fixed-size hash function.

2) 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%?

Step 1: Compute the probability that no one shares Trudy's birthday
There are 365 possible birthdays (ignoring leap years).
Each of the other N - 1 people has a 364 / 365 probability of not having the same birthday as Trudy.
The probability that none of the N - 1 people share Trudy's birthday is:

```
P(no match) = (364/365)^N - 1
```

Step 2: Compute the probability that at least one person shares Trudy's birthday

```
P(at least one match) = 1 - P(no match) = 1 - (364/365)^N - 1
```

Step 3: Solve for N where the probability exceeds 50%

```
1 - (364/365)^N - 1 > 0.5
= (364/365)^N - 1 < 0.5
= (N -1)ln(364/365) < ln(0.5) "taking the natural logarithm of both sides"
= Since ln(364/365) is negative -0.0027, dividing by it reverses the inequality: N -1 > (ln(0.5)/ln(364/365)
=  N -1 > (-0.6931/-0.0027)
≈ 252.66
Therefore, N >253.66
The smallest integer N that satisfies this condition is (254).
The minimum number of people for this probability to exceed 50% is 254.
```

3)In a room of N people (N ≤ 365), what's the probability of any two sharing a birthday, and what's the minimum N for this probability to be over 50%?

a - Probability calculation:

1st person: 365/365, 2nd person: 364/365, 3rd person: 363/365, continues until N-th person: (365 - N + 1)/365

P(all unique) = (365 x 364 x ... x (365-N+1)) / 365^N

```
P (all unique) = 365!/365^N(365 - N)!

P(at least one shared) = 1 - P(all unique)
```

b - Find the smallest N for which this probability exceeds 50%

1 - P(no shared birthday) > 0.5

by using calculation

```
N=1: P = 1 = 100% (trivially true)
N=10 P ≈ 0.883 (88.3%)
N=20 P ≈ 0.588 (58.8%)
N=23 P ≈ 0.493 (49.3%) First time it drops below 50%
```

```
we find that N=23 is the smallest number where the probability of at least one birthday match surpasses 50%.
```

By 70 people, the probability exceeds 99.9%

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

Hash functions create digital fingerprints, but their fixed-length design hides a critical flaw: unavoidable collisions.

Hash functions create digital fingerprints, but their fixed-length design hides a critical flaw: unavoidable collisions. The birthday attack exploits this by leveraging probability theory—instead of brute-forcing a specific hash match (requiring ~~2^n attempts for n-bit hashes), it hunts for any two colliding inputs using just ~√(total outputs) attempts (~~$2^{n/2}$). requiringThis clever shortcut makes finding collisions exponentially faster: while SHA-256 would need ~2^{256} guesses to reverse a specific hash, a birthday attack finds random collisions in ~2^{128} tries. That's why broken hashes like MD5 fell to this method, forcing modern systems to adopt longer outputs (SHA-3/512) that push collision searches beyond practical reach—turning a mathematical inevitability into a manageable security tradeoff.

5) Discuss the main issues associated with hash functions created using the Merkle-Damgård Construction process.

The Merkle-Damgård construction is a classic method for building cryptographic hash functions, used in algorithms like MD5, SHA-1, and SHA-256. but it has several well-known vulnerabilities and weaknesses:

a - Length Extension Attacks:

The Fatal Flaw: Given Hash(M), attackers can calculate Hash(M || padding || M') without knowing M—breaking authentication in naive MAC implementations

b - Collision-Finding Vulnerabilities:

Domino Effect: A single collision in the compression function extends to the full hash, as each block's output feeds into the next. This doomed MD5 and SHA-1

c - Predictable Padding:

Fixed Structure: Appending message length before processing creates patterns exploitable in chosen-prefix attacks.

d - Rigid Initialization:

Static IV: The fixed initial value (IV) lacks adaptability, making certain attacks easier if the compression function has weaknesses.

Part III: Code Project

Use jupyter notebook to do the coding problems.

1. Merkle Tree Implementation

   • Implement Merkle trees using SHA256 or some other hashing algorithms. Students can utilize existing crypto packages for hashing functions.

   • Each leaf node references a plaintext file.

   • Conduct a test with four leaf nodes, displaying the tree structure and hashes.

   • Conduct a second test with six leaf nodes, displaying the tree structure and hashes.

2. Root Hash Observation

   • Modify the content of one text file in the four-leaf-node scenario and compare the root hashes. Discuss your observations.

3. Hash Collision

   • Define a hash function using SHA256 but take only 4 bits as hash output.

   • Use the implementation in step 1(Merkle Tree Implementation) with this hash function.

   • Attempt to generate multiple text files with identical meanings but different hashes by altering file contents (e.g., adding spaces).

   • Find a hash collision among the text files. Discuss how many such files need to be generated. • Discuss strategies for finding collisions with hashes ranging from 4-bit to 160-bit in length.

4. Hash Puzzle

   • Using the chosen hashing algorithm (4 bits output), solve hash puzzles by finding hashes with a leading 1 zero bit, and then 2 zero bits.

   • Briefly discuss the workload involved in solving a puzzle requiring a 20-bit zero prefix for the SHA256.

Merkle Tree Implementation

• Implement Merkle trees using SHA256 or some other hashing algorithms. Students can utilize existing crypto packages for hashing functions.

• Each leaf node references a plaintext file.

• Conduct a test with four leaf nodes, displaying the tree structure and hashes.

• Conduct a second test with six leaf nodes, displaying the tree structure and hashes.

```
!pip install graphviz
```

```
Requirement already satisfied: graphviz in /usr/local/lib/python3.11/dist-packages (0.20.3)
```

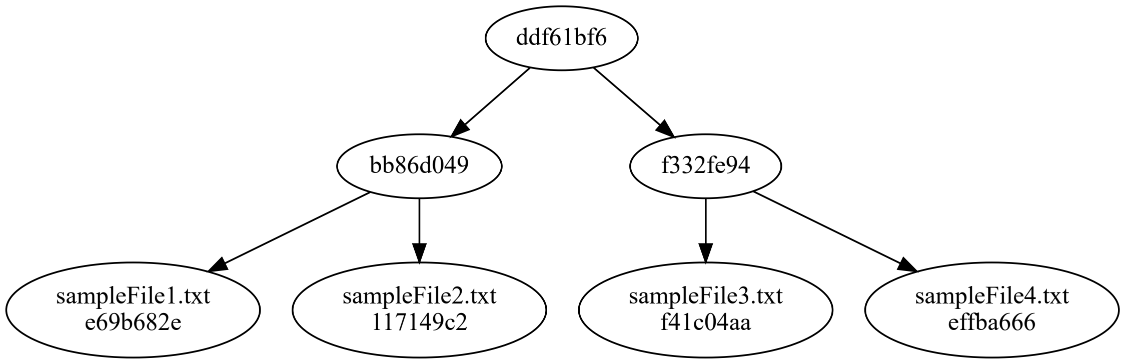
```
# Part 3 - Merkle Tree

# hashlib module implements a common interface for many secure cryptographic hash and message digest algorithms.
import hashlib
from graphviz import Digraph
from IPython.display import display, SVG

# Node class for Merkle tree
class Node:
    def __init__(self, left=None, right=None, hashValue=None, fileName=None):
        self.left = left
        self.right = right
        self.hash = hashValue  # store hash as bytes
        self.fileName = fileName  # non-null for leaf nodes

# Function to compute SHA256 hash of given data (in bytes)
def computeHash(data: bytes) -> bytes:
    return hashlib.sha256(data).digest()

# Build the Merkle tree from a list of file paths.
def buildMerkleTree(fileName):
    print('Here is file paths: ', fileName)
    # Create leaf nodes by reading file contents and computing their hashes.
    leaves = []
    for x in fileName:
        with open(x, 'rb') as f:
            data = f.read()
        hashData = computeHash(data)
        # print('Here is hash value: \n', hashData.hex())
        node = Node(hashValue=hashData, fileName=x)
        leaves.append(node)

    # Build the tree level by level until only the root remains.
    while len(leaves) > 1:
        # If the number of nodes is odd, duplicate the last node.
        if len(leaves) % 2 == 1:
            leaves.append(leaves[-1])
        temp = []
        # Pair nodes and compute parent node hash as SHA256(left.hash + right.hash)
        for i in range(0, len(leaves), 2):
            left = leaves[i]
            right = leaves[i+1]
            combined = left.hash + right.hash
            parentHash = computeHash(combined)
            parent = Node(left=left, right=right, hashValue=parentHash)
            temp.append(parent)
        leaves = temp
    return leaves[0]

# Recursively print the Merkle tree structure and hashes.
def printTree(node,level=0):
    if node.fileName:
        # This is a leaf node.
        print(f"{'    ' * level}Leaf - {node.fileName}: {node.hash.hex()}")
    else:
        # This is an internal node.
        if level == 0:
            print("Merkle Tree:")
            print(f"Root Node: {node.hash.hex()}:")
        else:
            print(f"{'    ' * level}Node: {node.hash.hex()}")
        if node.left:
            printTree(node.left, level+1)
        if node.right:
            printTree(node.right, level+1)

# Visualize the Merkle tree using graphviz
def visualizeTree(node):
    dot = Digraph()
    def addNodesEdges(node, parent=None):
        if node.fileName:
            node_id = node.fileName
            label = f"{node.fileName}\n{node.hash.hex()[:8]}"
        else:
            node_id = node.hash.hex()
            label = node.hash.hex()[:8]
        dot.node(node_id, label)
        if parent:
            dot.edge(parent, node_id)
        if node.left:
            addNodesEdges(node.left, node_id)
        if node.right:
            addNodesEdges(node.right, node_id)
    addNodesEdges(node)
    svg_data = dot.pipe(format='svg')
    display(SVG(svg_data))  # Display the SVG directly

# Create sample files based on the object key and values.
def savefile(fileObject):
    for fileName, content in fileObject.items():
        with open(fileName, 'w') as f:
            f.write(content)
        print("File saved: ", fileName);
```

1.1 - Each leaf node references a plaintext file.

Generate 4 text files with different content each

```
# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Sample Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4."
}

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree = buildMerkleTree(list(fileData.keys()))
print('\n')
printTree(merkTree)
# Visualize the Merkle tree
visualizeTree(merkTree)
```

```
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt
Here is file paths:  ['sampleFile1.txt', 'sampleFile2.txt', 'sampleFile3.txt', 'sampleFile4.txt']


Merkle Tree:
Root Node: ddf61bf634b0b650ff4441dca2478a126789aabbce062457dc4dbc8b24534bee:
    Node: bb86d04919c8fb4b512041626ecbc06a1cc91e0af6265cfddd3f3e3e64dd79f7
        Leaf - sampleFile1.txt: e69b682ec3a1cd190e86199720629550620d04a336c02920bfa2359e7ba1c624
        Leaf - sampleFile2.txt: 117149c26ba55ff6cccf913baf2ea8cfb9465b22d74409d3b2ecc8a067d12878
    Node: f332fe94cfba9b4ab96482730e79adbbee0e2c305a69e01801adbc2c5a913fe1
        Leaf - sampleFile3.txt: f41c04aa040bdd296ea18fd6e978812d741b619f2197571db003755aa720ce6a
        Leaf - sampleFile4.txt: effba6660f26c0ebdb2d5906f52f8d608e09071495489804204fa295c1ec3bd3
```

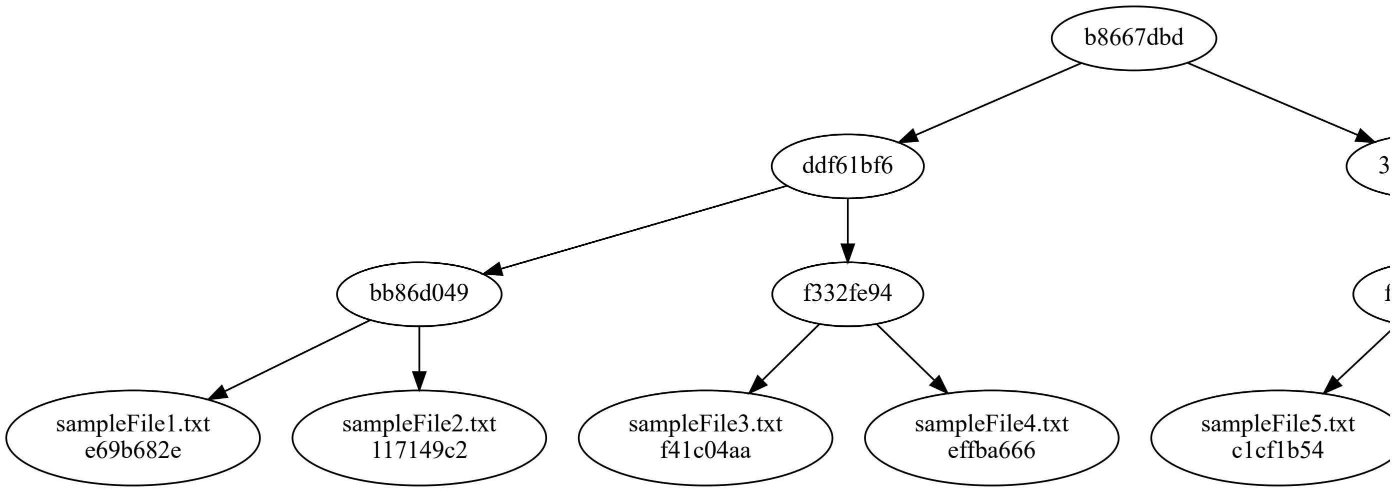


1.2 - Conduct a second test with six leaf nodes, displaying the tree structure and hashes.

Generate 6 text files with different content each. Repeating the last file to balance the tree. This was reccomended by the professor.

```
# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Sample Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4.",
    "sampleFile5.txt": "Sample Data 5.",
    "sampleFile6.1.txt": "Sample Data 6.",
    "sampleFile6.2.txt": "Sample Data 6.",
    "sampleFile6.3.txt": "Sample Data 6."
}

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree = buildMerkleTree(list(fileData.keys()))
print('\n')
printTree(merkTree)
visualizeTree(merkTree)
```

```
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt
File saved:  sampleFile5.txt
File saved:  sampleFile6.1.txt
File saved:  sampleFile6.2.txt
File saved:  sampleFile6.3.txt
Here is file paths:  ['sampleFile1.txt', 'sampleFile2.txt', 'sampleFile3.txt', 'sampleFile4.txt', 'sampleFile5.txt', 'sampleFile6.1.txt'


Merkle Tree:
Root Node: b8667dbdceaae9ec538262937ff58e77f40e5f9f5cd5a97610ebf1b461b9fec0:
    Node: ddf61bf634b0b650ff4441dca2478a126789aabbce062457dc4dbc8b24534bee
        Node: bb86d04919c8fb4b512041626ecbc06a1cc91e0af6265cfddd3f3e3e64dd79f7
            Leaf - sampleFile1.txt: e69b682ec3a1cd190e86199720629550620d04a336c02920bfa2359e7ba1c624
            Leaf - sampleFile2.txt: 117149c26ba55ff6cccf913baf2ea8cfb9465b22d74409d3b2ecc8a067d12878
        Node: f332fe94cfba9b4ab96482730e79adbbee0e2c305a69e01801adbc2c5a913fe1
            Leaf - sampleFile3.txt: f41c04aa040bdd296ea18fd6e978812d741b619f2197571db003755aa720ce6a
            Leaf - sampleFile4.txt: effba6660f26c0ebdb2d5906f52f8d608e09071495489804204fa295c1ec3bd3
    Node: 310eb95279582f0695867699676e54b60765ce15c3f57094e08159b4f8f8f41f
        Node: ff3b82f6dbd9cc55fa699a78fea5ae453eeba3ec97a6aba110f1fefcdb2c4b2f
            Leaf - sampleFile5.txt: c1cf1b540d950854698926e2ec1c23d34cf4e9bbc213814f9184c4709dd5230a
            Leaf - sampleFile6.1.txt: 5dd40139c92796738b7f41eb665c67d6807958ab0aead620143778b0fc8d2dad
        Node: dafe51e5ba8a7f06b2a2ad703d41968cddc605a6de1c2afa19154d077806f1e2
            Leaf - sampleFile6.2.txt: 5dd40139c92796738b7f41eb665c67d6807958ab0aead620143778b0fc8d2dad
            Leaf - sampleFile6.3.txt: 5dd40139c92796738b7f41eb665c67d6807958ab0aead620143778b0fc8d2dad
```



2 - Root Hash Observation

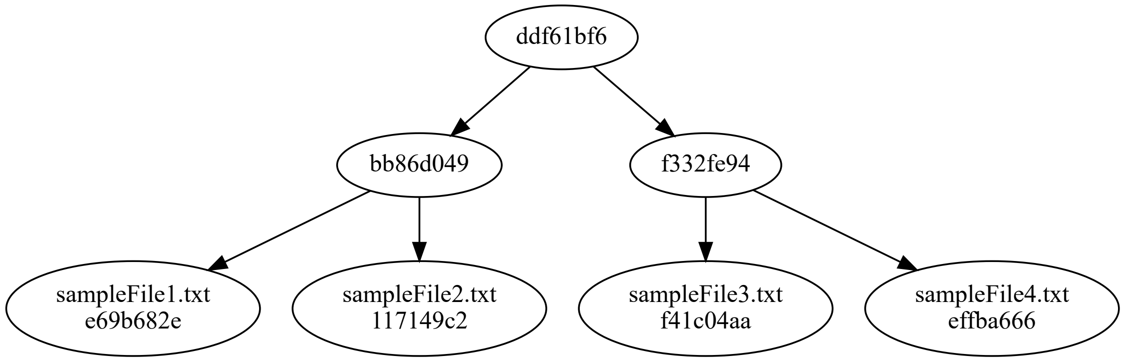
Modify the content of one text file in the four-leaf-node scenario and compare the root hashes. Discuss your observations.

Generate 4 text files with file 2 haveing different content than the first generation

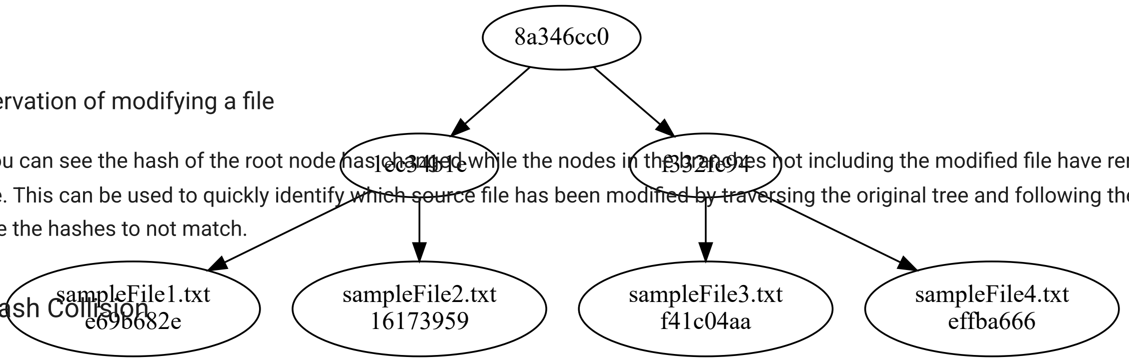
```python
# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Sample Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4.",
}

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree_orig = buildMerkleTree(list(fileData.keys()))

# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Different Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4.",
}

savefile(fileData)
```

```

# Build the Merkle tree with the fileData Object keys(Name)
merkTree_modified = buildMerkleTree(list(fileData.keys()))

print('\nOriginal Tree\n')
printTree(merkTree_orig)
print("\nModified File 2 Tree\n")
printTree(merkTree_modified)
print('\nOriginal Tree\n')
visualizeTree(merkTree_orig)
print("\nModified File 2 Tree\n")
visualizeTree(merkTree_modified)
```

```
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt
Here is file paths:  ['sampleFile1.txt', 'sampleFile2.txt', 'sampleFile3.txt', 'sampleFile4.txt']
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt
Here is file paths:  ['sampleFile1.txt', 'sampleFile2.txt', 'sampleFile3.txt', 'sampleFile4.txt']

Original Tree

Merkle Tree:
Root Node: ddf61bf634b0b650ff4441dca2478a126789aabbce062457dc4dbc8b24534bee:
    Node: bb86d04919c8fb4b512041626ecbc06a1cc91e0af6265cfddd3f3e3e64dd79f7
        Leaf - sampleFile1.txt: e69b682ec3a1cd190e86199720629550620d04a336c02920bfa2359e7ba1c624
        Leaf - sampleFile2.txt: 117149c26ba55ff6cccf913baf2ea8cfb9465b22d74409d3b2ecc8a067d12878
    Node: f332fe94cfba9b4ab96482730e79adbbee0e2c305a69e01801adbc2c5a913fe1
        Leaf - sampleFile3.txt: f41c04aa040bdd296ea18fd6e978812d741b619f2197571db003755aa720ce6a
        Leaf - sampleFile4.txt: effba6660f26c0ebdb2d5906f52f8d608e09071495489804204fa295c1ec3bd3

Modified File 2 Tree

Merkle Tree:
Root Node: 8a346cc014d5811cffbf8baaf09d9c8aefbd6a739d95182a879a1bd7f9b16411:
    Node: 1ec34b1e9a37118e899b389eb0a1b4742b8a7ad39828dbecff7a483413792707
        Leaf - sampleFile1.txt: e69b682ec3a1cd190e86199720629550620d04a336c02920bfa2359e7ba1c624
        Leaf - sampleFile2.txt: 161739594b36f554dd0031d208eaef02cb3f69e8a0a3091cf08487bc4d62eab5
    Node: f332fe94cfba9b4ab96482730e79adbbee0e2c305a69e01801adbc2c5a913fe1
        Leaf - sampleFile3.txt: f41c04aa040bdd296ea18fd6e978812d741b619f2197571db003755aa720ce6a
        Leaf - sampleFile4.txt: effba6660f26c0ebdb2d5906f52f8d608e09071495489804204fa295c1ec3bd3

Original Tree
```



```
Modified File 2 Tree
```



Obvervation of modifying a file

As you can see the hash of the root node has changed while the nodes in the branches not including the modified file have remained the same. This can be used to quickly identify which source file has been modified by traversing the original tree and following the branches where the hashes to not match.

3. Hash Collision

• Define a hash function using SHA256 but take only 4 bits as hash output.

• Use the implementation in step 1(Merkle Tree Implementation) with this hash function.

• Attempt to generate multiple text files with identical meanings but different hashes by altering file contents (e.g., adding spaces).

• Find a hash collision among the text files. Discuss how many such files need to be generated.

• Discuss strategies for finding collisions with hashes ranging from 4-bit to 160-bit in length.

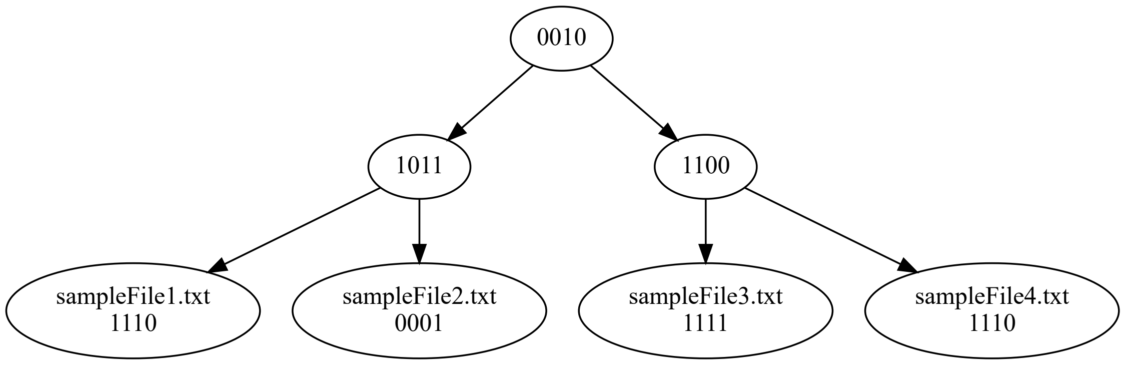
3.1 & 3.2

- Define a hash function using SHA256 but take only 4 bits as hash output.
- Use the implementation in step 1(Merkle Tree Implementation) with this hash function.

```python
# Part 3.1 & 3.2 - Merkle Tree with Optional bit size setting

# hashlib module implements a common interface for many secure cryptographic hash and message digest algorithms.
import hashlib
from graphviz import Digraph
from IPython.display import display, SVG

# Node class for Merkle tree
class Node:
    def __init__(self, left=None, right=None, hashValue=None, fileName=None):
        self.left = left
        self.right = right
        self.hash = hashValue  # store hash as bytes
        self.fileName = fileName  # non-null for leaf nodes

# Function to compute SHA256 hash of given data (in bytes) but defined the number of bits to be used.
def computeHash4Bits(data: bytes) -> str:
    """Computes the SHA256 hash and returns the most significant 4 bits as a binary string."""
    full_hash = hashlib.sha256(data).digest()

    # Get the most significant 4 bits from the first byte
    most_significant_4_bits = (full_hash[0] >> 4) & 0x0F

    # Convert to binary string representation
    binary_string = f'{most_significant_4_bits:04b}'

    return binary_string

# Build the Merkle tree from a list of file paths.
def buildMerkleTree4Bits(fileName):
    # Create leaf nodes by reading file contents and computing their hashes.
    leaves = []
    for x in fileName:
        with open(x, 'rb') as f:
            data = f.read()
        hashData = computeHash4Bits(data)
        # print('Here is hash value: \n', hashData.hex())
        node = Node(hashValue=hashData, fileName=x)
        leaves.append(node)

    # Build the tree level by level until only the root remains.
    while len(leaves) > 1:
        # If the number of nodes is odd, duplicate the last node.
        if len(leaves) % 2 == 1:
            leaves.append(leaves[-1])
        temp = []
        # Pair nodes and compute parent node hash as SHA256(left.hash + right.hash)
        for i in range(0, len(leaves), 2):
            left = leaves[i]
            right = leaves[i+1]
            combined = (left.hash + right.hash).encode()
            parentHash = computeHash4Bits(combined)
            parent = Node(left=left, right=right, hashValue=parentHash)
            temp.append(parent)
        leaves = temp
    return leaves[0]

# Recursively print the Merkle tree structure and hashes.
def printTree(node,level=0):
    if node.fileName:
        # This is a leaf node.
        print(f"{'    ' * level}Leaf - {node.fileName}: {node.hash}")
    else:
        # This is an internal node.
        if level == 0:
            print("Merkle Tree:")
            print(f"Root Node: {node.hash}:")
        else:
            print(f"{'    ' * level}Node: {node.hash}")
        if node.left:
            printTree(node.left, level+1)
        if node.right:
            printTree(node.right, level+1)

# Visualize the Merkle tree using graphviz
def visualizeTree(node):
    dot = Digraph()
    def addNodesEdges(node, parent=None):
        if node.fileName:
            node_id = node.fileName
            label = f"{node.fileName}\n{node.hash[:8]}"
        else:
            node_id = node.hash
            label = node.hash[:8]
        dot.node(node_id, label)
        if parent:
            dot.edge(parent, node_id)
        if node.left:
            addNodesEdges(node.left, node_id)
        if node.right:
            addNodesEdges(node.right, node_id)
    addNodesEdges(node)
    svg_data = dot.pipe(format='svg')
    display(SVG(svg_data))  # Display the SVG directly

# Create sample files based on the object key and values.
def savefile(fileObject):
    for fileName, content in fileObject.items():
        with open(fileName, 'w') as f:
            f.write(content)
        print("File saved: ", fileName);
```

```
# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Sample Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4."
}

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree = buildMerkleTree4Bits(list(fileData.keys()))
print('\n')
printTree(merkTree)
# Visualize the Merkle tree
visualizeTree(merkTree)
```

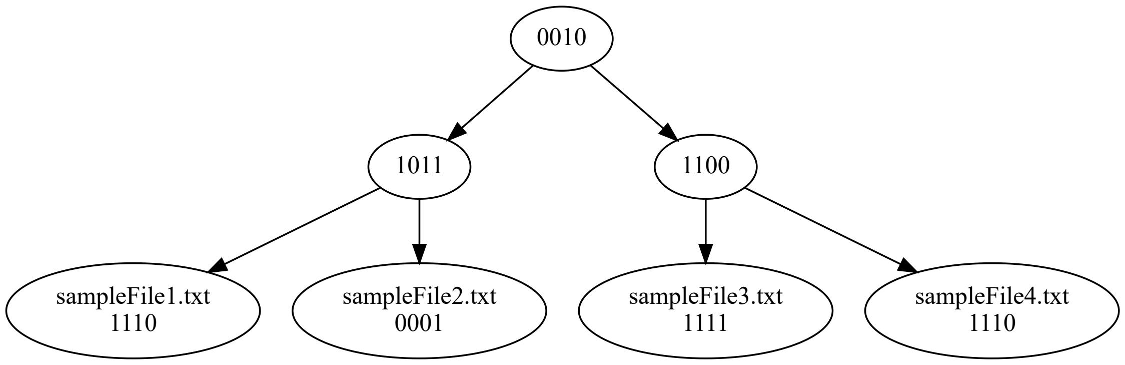
```
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt


Merkle Tree:
Root Node: 0010:
    Node: 1011
        Leaf - sampleFile1.txt: 1110
        Leaf - sampleFile2.txt: 0001
    Node: 1100
        Leaf - sampleFile3.txt: 1111
        Leaf - sampleFile4.txt: 1110
```



3.2 4 Bit Hash

```
# Define file names and contents
fileData = {
    "sampleFile1.txt": "Sample Data 1.",
    "sampleFile2.txt": "Sample Data 2.",
    "sampleFile3.txt": "Sample Data 3.",
    "sampleFile4.txt": "Sample Data 4."
}

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree = buildMerkleTree4Bits(list(fileData.keys()))
print('\n')
printTree(merkTree)
# Visualize the Merkle tree
visualizeTree(merkTree)
```

```
File saved:  sampleFile1.txt
File saved:  sampleFile2.txt
File saved:  sampleFile3.txt
File saved:  sampleFile4.txt


Merkle Tree:
Root Node: 0010:
    Node: 1011
        Leaf - sampleFile1.txt: 1110
        Leaf - sampleFile2.txt: 0001
    Node: 1100
        Leaf - sampleFile3.txt: 1111
        Leaf - sampleFile4.txt: 1110
```



3.3 Attempt to generate multiple text files with identical meanings but different hashes by altering file contents (e.g., adding spaces).

```
# Define file names and contents
fileData = {}
for i in range(4):
  contents = f"Sample Data {' ' * i}."
  fileData[f"collisionFile{i}.txt"] = contents

savefile(fileData)

# Build the Merkle tree with the fileData Object keys(Name)
merkTree = buildMerkleTree4Bits(list(fileData.keys()))
print('\n')
printTree(merkTree)
# Visualize the Merkle tree
visualizeTree(merkTree)
```

```
File saved:  collisionFile0.txt
File saved:  collisionFile1.txt
File saved:  collisionFile2.txt
File saved:  collisionFile3.txt

Merkle Tree:
Root Node: 1101:
    Node: 1111
        Leaf - collisionFile0.txt: 1100
        Leaf - collisionFile1.txt: 0000
```

3.4 Find a hash collision among the text files using 4 bit hashes.

Discuss how many such files need to be generated.

(In liu of storing the data in files, the string contents will be hashed to save space on disk.)

```python
# given some contents, continiously append a space to the contentes until a collision is found
def find_collision(contents:str, bits:int = 4):
  hashes = {}

  # Generating 3 different example cases
  while True:
    contents = contents + " "
    hash = computeHash4Bits(contents.encode())
    if hash in hashes:
      break
    hashes[hash] = contents

  hash_collision = len(hashes) + 1
  # print(f'The number of files needed to generate a collision is: {hash_collision}')
  return hash_collision

collision_counts = []
trials = 100
bits = 4
for i in range(trials):
  collision_counts.append(find_collision(f"Sample Data {i}.", bits))

print(f'After {trials} trials using {bits} bit hashes, the average number of files needed to generate a collision is: {sum(collision_cou
```

```
After 100 trials using 4 bit hashes, the average number of files needed to generate a collision is: 5.84
```

Since a 4 bit hash only has 16 different values, the number of files needed to generate a collision is slightly less than 50% at 5.84 after 100 trials.

3.5 Discuss strategies for finding collisions with hashes ranging from 4-bit to 160-bit in length.

For the 4 bit collision check at 2^4 or 16 different values, a brute force option is possible. When we extend the bits to 160, this becomes more complicated since the number of values that can be generated in 160 bits is enormous. At 160 bites (2^160) or we get 1.46 × 10^48 different values. Even at 50%, 2^80 or 1.2 × 10^24, is quite large number and brute forcing would take considerable time.

4.0 Hash Puzzle

- Using the chosen hashing algorithm (4 bits output), solve hash puzzles by finding hashes with a leading 1 zero bit, and then 2 zero bits.
- Briefly discuss the workload involved in solving a puzzle requiring a 20-bit zero prefix for the SHA256.

a - hashing algorithm (4 bits output), solve hash puzzles by finding hashes with a leading 1 zero bit, and then 2 zero bits.

```python
import hashlib

def computeHash4Bits(data):
    # First compute a standard hash (SHA-256)
    full_hash = hashlib.sha256(data).hexdigest()
    # Then take just the first character (4 bits) of the hex representation
    first_char = full_hash[0]
    # Convert to binary (4 bits) and remove the '0b' prefix
    binary_4bits = bin(int(first_char, 16))[2:].zfill(4)
    return binary_4bits
```