Lab 01
Basic Hash Functions and File Integrity Checking

Learning Outcomes

By the end of this lab, students will be able to

1. Explain what hash functions are and how they are used in file integrity checking.
2. Demonstrate how to calculate hash values using both Windows and Linux tools.
3. Analyze how hash functions perform with different file sizes.
4. Explain why hash functions are sensitive to small changes in input.

Background

Hash function help protect data by creating a unique code (called a hash value) from any input, like a file or text. If someone changes even one small part of the input, the hash value changes completely. This is called the avalanche effect.

Important characteristics of hash functions:

- Deterministic: The same input always gives the same output.
- One-way: It's very hard to find the original input from the hash value.
- Sensitive to Changes: Even a tiny change in the input leads to a big change in the hash value.

Applications of Hash functions:

- File Integrity and Checksums: Hash functions are widely used to verify the integrity of files during transmission or storage. By generating a hash (also know as a checksum) for the original file and comparing it to the received file, you can quickly detect any changes or corruption. This is commonly used in software downloads and updates to ensure files are not tempered with.

- Password Security: In many systems, passwords are hashed and stored as hash values rather than plain text. When a user logs in, the entered password is hashed, and the system compares the resulting hash with the stored hash. This process protects passwords, as the original data cannot easily be retrieved from the hash.
- Digital Signatures and Message Authentication: Hash functions are crucial for verifying the authenticity and integrity of messages and documents through digital signatures and Message Authentication Codes (MACs). By hashing the content and signing or encrypting it with a private key, systems ensure that data has not been altered and confirm the identity of the sender. This is commonly used in secure communications and electronic document signing.
- Blockchain and Cryptocurrencies: In blockchain technology, hash functions help link blocks securely by including the hash of the previous block in the new block. This ensures that altering any block would break the chain. Cryptocurrencies like Bitcoin also rely on hash functions for validating transactions and mining new blocks.
- Data deduplication and Anti-Virus Detection: Hash functions help in detecting duplicate data in storage systems (by comparing hash values) and identifying malware in anti-virus software. Each file or malware generates a unique hash value, allowing systems to quickly identify duplicate files or detect known malicious files by comparing their hashes to a database.
- Digital Forensics: Hash functions are used in digital forensics to verify that digital evidence (e.g., file, hard drives) has not been altered. Investigators generate a hash of the original data and regularly compare it to ensure the evidence remains unchanged throughout the investigation process.

Commonly Used Hash functions:

- MD5: A 128-bit hash function that was once widely used for file integrity checks, but is now considered insecure due to its vulnerability to collision attacks.
- SHA-1: A 160-bit hash function that was used for verifying data integrity and digital signatures, but is also now deprecated due to security weakness.

- SHA-256: A 256-bit hash function from the SHA-2 family, widely used in security applications such as blockchain, SSL/TLS certificates, and password hashing. It offers strong collision resistance and is currently considered secure.

Links for further reading

- https://www.thesslstore.com/blog/what-is-a-hash-function-in-cryptography-a-beginners-guide/
- http://www.unixwiz.net/techtips/iguide-crypto-hashes.html

Equipment

- Windows Computer with PowerShell installed or a Windows Virtual Machine.
- Linux Computer with terminal access (e.g., Ubuntu) or a Linux Virtual Machine.
- GtkHash installed on either Windows or Linux (Download from https://gtkhash.org/).
- VirtualBox (if using virtual machines):
    - Download from https://www.virtualbox.org/
    - For Windows 11 Virtual Machine:
        - You can download a pre-configured Windows11 Virtual Machine Appliance from https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/
    - For Ubuntu Linux Virtual Machine:
        - You can download a pre-configured Ubuntu Linux VirtualBox Disk Image from https://www.osboxes.org/ubuntu/
        - Alternatively, you can perform a fresh installation using the Ubuntu installer disk image, available at https://ubuntu.com/download/desktop

Lab Procedures

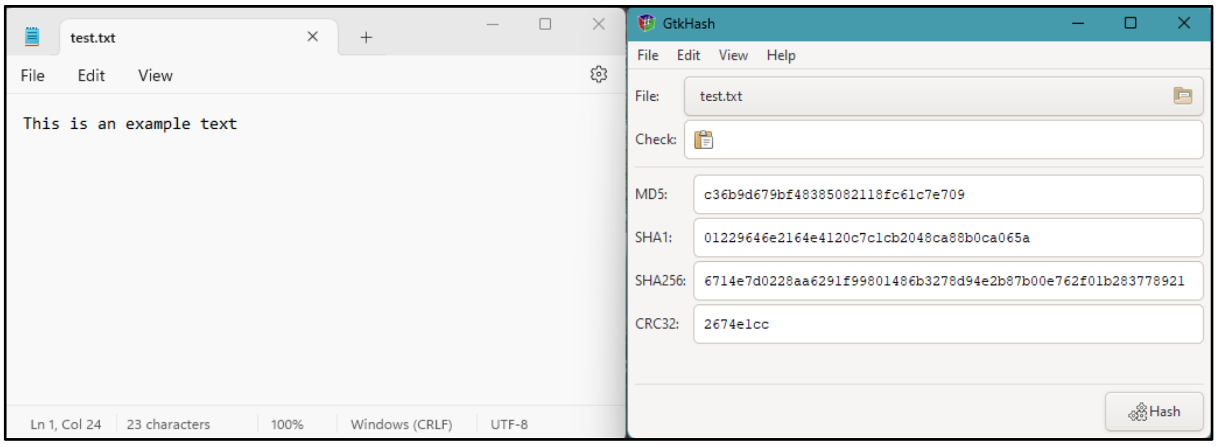
Experiment 1.1: File Integrity Checking Using GtkHash

Objective:

- To calculate and compare hash values of a text file using different algorithms (MD5, SHA-1, and SHA-256) using program GtkHash either in Windows or Linux, and observe how changes in the file affect the hash values.

Instructions for Windows

1. Download and Install GtkHash
   - Download GtkHash from https://gtkhash.org/ and follow the installation instructions.
2. Create a Text File
   - Open a text editor (e.g. Notepad) and create a new file.
   - Write a few sentences in the file (e.g., “This is a test file for hashing.”)
   - Save the file as ‘test.txt’ on your desktop or another location you can easily access.
3. Compute Hash Values
   - Open GtkHash and select ‘test.txt’. Compute the hash values using MD5, SHA-1, and SHA-256
   - Write down or screen shot the generated hash values for each algorithm as shown in Figure 1



```
This is an example text
```

```
File:   test.txt
Check:
MD5:    c36b9d679bf48385082118fc61c7e709
SHA1:   01229646e2164e4120c7c1cb2048ca88b0ca065a
SHA256: 6714e7d0228aa6291f99801486b3278d94e2b87b00e762f01b283778921
CRC32:  2674e1cc
```

Figure 1 - Example Hash value recording

4. Modify the Text File
   - Open ‘test.txt’ in Notepad again.
   - Modify the text, for example, change it to “This is a modified test file for hashing.”
   - Save the file.

5. Recompute Hash Values
    - Use GtkHash to recompute the hash values of the modified file.
    - Record or screenshot the new hash values.
6. Compare the Hash Values
    - Compare the original and modified hash values.
    - Note the difference and explain how even small changes to the text result in completely different hash values (avalanche effect).

Instructions for Linux (Ubuntu)

1. Download and Install GtkHash
    - Open the terminal and run the following command to install GtkHash

```
sudo apt install gtkhash
```

    - Once installed, verify by typing ‘gtkhash’ in the terminal, or open it from the Applications menu.
2. Create a Text File
    - Open the terminal and use the ‘nano’ command to create a file

```
nano test.txt
```

    - Type some text, for example “This is a test file for hashing.”
    - Press ‘Ctrl + x’ then ‘y’, and ‘Enter’ to save the file and exit ‘nano’.
3. Compute Hash Values Using GtkHash
    - Open GtkHash from the terminal by typing ‘gtkhash’ or launch it from the Applications menu.
    - In GtkHash browse and select the ‘test.txt’ file you created.
    - Click the Hash button to generate the hash values
    - Write down or screenshot the generated hash values for MD5, SHA-1, and SHA-256
4. Modify the Text File
    - In the terminal, open the file for editing using ‘nano’
    - Modify the text, for example, change it to “This is a modified test file for hashing.”
    - Press ‘Ctrl + x’, then ‘y’, and ‘Enter’ to save the file and exit ‘nano’
5. Recompute Hash Values
    - Open GtkHash again and select the modified ‘test.txt’ file.
    - Recompute the hash values
    - Record or screenshot the new hash values.
6. Compare the Hash Values

- Compare the original and modified hash values
- Note the difference and explain how even small changes to the text result in completely different hash values (avalanche effect).

Experiment 1.2: Hashing with Windows PowerShell

Objective:

- To compute hash values for a text file using the built-in Windows PowerShell commands and observe how changes to file affect the hash values.

Instructions

1. Create a Text File
   - Open Notepad.
   - Type some text, for example, “This is a test file for hashing.”
   - Save the file as ‘test.txt’ on your desktop.
2. Open Windows PowerShell
   - Press ‘Win + x’ and select “Windows PowerShell” from the menu.
   - PowerShell will open in the default directory (usually C:\Users\<username>)
3. Navigate to the File Location
   - Use the ‘cd’ command to navigate to the directory where ‘test.txt’ is saved.
4. Compute Hash Values
   - Use the following PowerShell commands to compute the hash values for MD5, SHA-1, and SHA-256
     - MD5

```
Certutil -hashfile test.txt MD5
```

     - SHA-1

```
Certutil -hashfile test.txt SHA1
```

     - SHA-256

```
Certutil -hashfile test.txt SHA256
```

   - Write down or screenshot the hash values for MD5, SHA-1, and SHA-256
5. Modify the Text File
   - Open ‘test.txt’ in Notepad again
   - Modify the text, for example, change it to “This is a modified test file for hashing.”
   - Save the file

6. Recompute Hash values in PowerShell
   - In PowerShell, re-run the same hash command for the modified ‘test.txt’ file
   - Write down or screenshot the new hash values for the modified file.

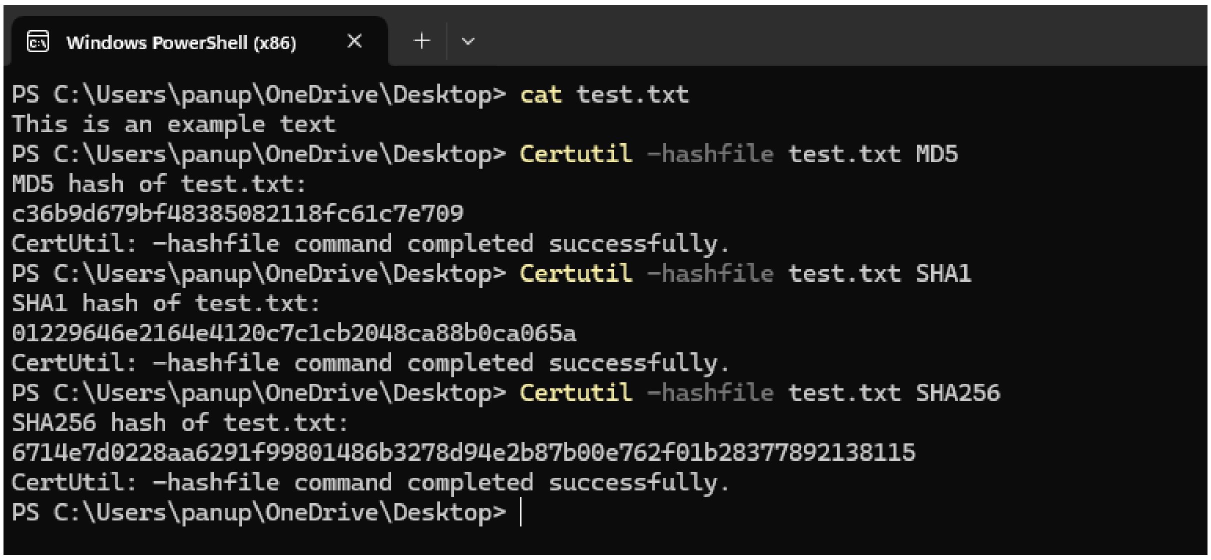

```
PS C:\Users\panup\OneDrive\Desktop> cat test.txt
This is an example text
PS C:\Users\panup\OneDrive\Desktop> Certutil -hashfile test.txt MD5
MD5 hash of test.txt:
c36b9d679bf48385082118fc61c7e709
CertUtil: -hashfile command completed successfully.
PS C:\Users\panup\OneDrive\Desktop> Certutil -hashfile test.txt SHA1
SHA1 hash of test.txt:
01229646e2164e4120c7c1cb2048ca88b0ca065a
CertUtil: -hashfile command completed successfully.
PS C:\Users\panup\OneDrive\Desktop> Certutil -hashfile test.txt SHA256
SHA256 hash of test.txt:
6714e7d0228aa6291f99801486b3278d94e2b87b00e762f01b28377892138115
CertUtil: -hashfile command completed successfully.
PS C:\Users\panup\OneDrive\Desktop> |
```

Figure 2 Example Hash value from PowerShell recording

7. Compare the Hash Values
   - Compare the original hash values with the new ones after modifying the text.
   - Explain how even small changes in the file resulted in different hash values for MD5, SHA-1, and SHA-256 (avalanche effect).

<u>Experiment 1.3</u>: Hashing with Linux Terminal

<u>Objective</u>:
- To compute hash values for a text file using Linux terminal commands and observe how changes to the file affect the hash values.

<u>Instructions</u>

1. Create a Text File
   - Open the Linux Terminal by pressing ‘Ctrl + Alt + t’
   - Use the ‘nano’ command to create and open a new text file
2. Navigate to the File Location
   - If the file is saved in another directory (e.g., the Desktop), use the ‘cd’ (change directory) command to navigate to that location
3. Compute Hash Values in Linux
   - Use the following commands to compute the hash values for MD5, SHA-1, and SHA-256

- MD5

```
md5sum test.txt
```

- SHA-1

```
sha1sum test.txt
```

- SHA-256

```
sha256sum test.txt
```

4. Modify the Text File
   - Open the ‘test.txt’ file again using ‘nano’
   - Modify the text, for example, change it to “This is a modified test file for hashing.”
   - Press ‘Ctrl+x’, then ‘y’, and hit ‘Enter’ to save the file and exit.
5. Recompute Hash Values in Linux
   - Re-run the same commands to compute the hash values for the modified file
6. Compare the Hash Values
   - Compare the original hash values with the new ones after modifying the text.
   - Explain how even small changes in the file resulted in different hash values for MD5, SHA-1, and SHA-256 (avalanche effect).

Experiment 1.4: Performance analysis of Hashing

Objective:

- To compute hash values for files of varying sizes using a selected hash algorithm (MD5, SHA-1, and SHA-256) on Linux. Students will create test files using a shell script and measure the time taken to compute the hash for each file, analyzing the performance based on file size.

Important Warning

Running script in this experiment will generate large files. Please ensure your system has sufficient disk space before running the script. The process may take significant time, especially for large files, depending on your system performance.

Instructions

1. Use the Shell Script to Create Test Files
    - Write the shell script file using nano

```
nano create_test_files.sh
```

- Save the script as shown in Listing 1 'create_test_files.sh'
- You can download script from https://gitfront.io/r/panupongs/jLnSRnrD8qtR/Security-Lab-GitFront/
- Make the script Executable using 'chmod' command

```
chmod +x create_test_files.sh
```

- Run the Script

```
./create_test_files.sh
```

- Verify the Created Files by listing the files in the directory

```
ls -lh
```

- You should see files named file_1KB.txt, file_512MB.txt, file_1GB.txt, etc. , with the correct sizes then capture the screen to record results.

2. Test Hash Performance and Collect Results
    - Write the shell script file using nano

```
nano create_test_files.sh
```

- Save the script as shown in Listing 2 'hash_test.sh'
- You can download the script from https://gitfront.io/r/panupongs/jLnSRnrD8qtR/Security-Lab-GitFront/
- Make the Script Executable

```
chmod +x hash_test.sh
```

- Run the Script

```
./hash_test.sh
```

- Choose a Hash Algorithm

- After completing the test, a log file will be generated with the results for the selected algorithm.

3. Repeat the process for the Other Algorithms
    - Run the script again
    - Select a Different Algorithm
    - Running the script for each algorithm until all three algorithms (MD5, SHA-1, and SHA-256) have been tested.
4. Analyze the Results
    - Open the log Files (hash_results_MD5.log, hash_results_SHA-1.log, hash_results_SHA-256.log)
    - Extract the data (average time for each hash algorithm and each size of test file from the log files.)
    - Create a table of results. The table should list: file size, average MD5 time, average SHA-1 time, and Average SHA-256 time.
    - Plot the graph of average time versus file size for each algorithm.
    - Compare the Performance of Each Algorithm
    - Identify Trends

Listing 1: create_test_files.sh

```bash
#!/bin/bash

# Ensure pv (Pipe Viewer) is installed for progress diplay
if ! command -v pv &> /dev/null
then
 echo "The 'pv' utility is not installed. Installing it now..."
 sudo apt install -y pv
fi

# Declare file sizes with labels for Set1 (KB), Set2 (MB), and Set3 (GB)
sizes_set1=("1KB:1024" "2KB:2048" "4KB:4096" "8KB:8192"
 "16KB:16384" "32KB:32768" "64KB:65536" "128KB:131072"
 "256KB:262144" "512KB:524288")

sizes_set2=("1MB:1048576" "2MB:2097152" "4MB:4194304"
 "8MB:8388608" "16MB:16777216" "32MB:33554432" "64MB:67108864"
 "128MB:134217728" "256MB:268435456" "512MB:536870912")


sizes_set3=("1GB:1073741824" "2GB:2147483648" "3GB:3221225472"
 "4GB:4294967296" "5GB:5368709120" "6GB:6442450944" "7GB:7516192768"
 "8GB:8589934592" "9GB:9663676416" "10GB:10737418240")


# Function to create files based on the sizes
create_files(){
 local -n size_list=$1
 for size_label in "${size_list[@]}"
 do
     label=$(echo "$size_label" | cut -d':' -f1)
     size=$(echo "$size_label" | cut -d':' -f2)
     filename="file_${label}.txt"
     echo "Creating $filename with size $label ..."

     # Generate the file using random base64-encoded data
     base64 /dev/urandom | pv -s "$size" | head -c "$size" > "$filename"
     echo "$filename created."
 done
}

# Create files for Set1 (1KB to 512KB)
echo "Creating files for Set1 (1KB to 512KB)..."
create_files sizes_set1
# Create files for Set2 (1MB to 512MB)
echo "Creating files for Set2 (1MB to 512MB)..."
create_files sizes_set2
# Create files for Set3 (1GB to 10GB)
create_files sizes_set3

echo "All test files created successfully!
```

Listing 2 hash_test.sh

```
#!/bin/bash
# Declare file sizes with labels for set1 (KB), set2 (MB), and set3 (GB)
sizes_set1=("1KB" "2KB" "4KB" "8KB" "16KB" "32KB" "64KB" "128KB" "256KB" "512KB")
sizes_set2=("1MB" "2MB" "4MB" "8MB" "16MB" "32MB" "64MB" "128MB" "256MB" "512MB")
sizes_set3=("1GB" "2GB" "3GB" "4GB" "5GB" "6GB" "7GB" "8GB" "9GB" "10GB")

# Prompt user to select hash algorithm
echo "Select hash algorithm to test:1) MD5 2) SHA-1 3) SHA-256"
read -p "Enter the number (1, 2, or 3): " algo_choice

# Set algorithm and log file name based on user choice
case "$algo_choice" in
 1)algo_name="MD5"   algo_cmd="md5sum" ;;
 2)algo_name="SHA-1" algo_cmd="sha1sum";;
 3)algo_name="SHA-256"   algo_cmd="sha256sum";;
 *)echo "Invalid choice. Please choose 1, 2, or 3." exit 1 ;;
esac

# log file to store hash results
log_file="hash_results_${algo_name}.log"
# clear the log file if it exists
> "$log_file"

# Function to hash files three times and calculate average time
hash_files(){
  local -n size_list=$1
  for label in "${size_list[@]}"; do
      filename="file_${label}.txt"
      if [[ -f "$filename" ]]; then
         echo "Hashing $filename with $algo_name ..." | tee -a "$log_file"
  total_time=0
  for i in {1..3}; do
     # Extract the real time (in seconds) for the hash computation
     real_time=$({ time "$algo_cmd" "$filename" >/dev/null;} 2>&1 | grep real | awk '{print $2}')
     minutes=$(echo "$real_time" | cut -d'm' -f1)
     seconds=$(echo "$real_time" | cut -d'm' -f2 | sed 's/s//')
     # Convert minutes to seconds and add to total
     total_time=$(echo "$total_time + ($minutes * 60) + $seconds" | bc)
          done
   # Calculate average time
   avg_time=$(echo "scale=3; $total_time / 3" | bc)
   echo "Average Time ($algo_name): $avg_time seconds" | tee -a "$log_file"
   echo "" | tee -a "$log_file" # Add an empty line for readability
        else echo "File $filename not found. Skipping..." | tee -a "$log_file"
        fi
  done
}

# Hash files for set1 (1KB to 512KB)
echo "Hashing files for Set1 (1KB to 512KB) using $algo_name ..." | tee -a "$log_file"
hash_files sizes_set1
# Hash files for set2 (1MB to 512MB)
echo "Hashing files for Set2 (1MB to 512MB) using $algo_name ..." | tee -a "$log_file"
hash_files sizes_set2
# Hash files for set3 (1GB to 10GB)
echo "Hashing files for set3 (1GB to 10GB) using $algo_name..." | tee -a "$log_file"
hash_files sizes_set3

echo "Hashing completed. Results recorded in $log_file"
```

Post-Lab Questions

For Experiment 1.1
1. If you rename a file but do not modify its contents, will the hash value change? Why or why not?
2. Compare the hash values generated by MD5, SHA-1, and SHA-256 in this experiment.
3. What happens to the hash value when you make even a small change in the file content (e.g., adding or removing a character)?

For Experiment 1.2
1. List the algorithms supported by CertUtil for hash computation.
2. Were the hash values calculated using CertUtil in PowerShell the same as those generated by GtkHash? Why or why not?
3. What advantages does CertUtil in PowerShell provided compared to using graphical tools like GtkHash?
4. Explain the syntax of the CertUtil command for hashing

For Experiment 1.3
1. Explain the process of calculating hash values in Linux using terminal commands
2. What differences, if any, did you observe between the hash values generated by Linux terminal commands and those generated by CertUtil in Windows or GtkHash?
3. In what situations would a system administrator prefer to use terminal-based hash commands over graphical tools?

For Experiment 1.4
1. What relationship did you find between file size and the time taken to compute the hash?
2. Which hash algorithm (MD5, SHA-1, or SHA-256) performed the fastest in your tests? Why do you think this was the case?
3. What real-world applications might require fast and efficient hashing of large file?
4. How does the performance of the three hash algorithms compare for the different file sizes?

General Questions for All Experiments
1. What role do hash functions play in ensuring the integrity and security of data in real-world applications?
2. How would you apply the knowledge from these experiments to real-world problems involving file integrity and security?